Automatic generation of visualization web services

Internship report by
Magali Lapeyre-Mirande

Université de Pau et des Pays de l’Adour
Master Technologies de l’Internet - 2nd year - June 2008
In the first place, I’d like to thank Dr. Gordon Erlebacher, professor at the School of Computational Science at Florida State University, for the confidence he trusted upon me as well as his support and advice during my internship.

Next, I’d like to express my appreciation for Yenan Qu and Evan Bollig, both working under the instruction of Dr. Erlebacher, for welcoming me in the project team and supporting me.

I’d also like to thank my two roommates and colleagues, Julien Lafourcade and Nicolas Masson, for their advice and daily good temper.

Finally, I’d like to express my appreciation for my parents and brother, who gave me great support and feedback throughout this internship, even if we were miles away.
## Contents

1 Introduction ............................................. 6

2 Presentation ............................................ 7

3 Related Work ............................................ 8
   3.1 Introduction to web services ......................... 8
      3.1.1 XML messaging ........................................ 8
      3.1.2 Discovery ............................................. 9
      3.1.3 Description ........................................... 9
   3.2 WATT .................................................. 9
   3.3 KWATT ................................................. 9
      3.3.1 Technologies used ................................... 9
      3.3.2 KWATT architecture ................................. 10
      3.3.3 Criticism ............................................ 11

4 Project Management ....................................... 13
   4.1 Project specifications ................................. 13
      4.1.1 Project goal ......................................... 13
      4.1.2 Specifications ....................................... 13
      4.1.3 Working environment ............................... 13
   4.2 Realization ............................................ 15
   4.3 Documentation ....................................... 15

5 Automatic services generation from Ruby scripts .... 16
   5.1 Technologies used ..................................... 16
   5.2 Modifications to KWATT architecture ................ 16
      5.2.1 Object-Oriented KWATT ......................... 17
      5.2.2 CMake configuration file generation ........... 18
   5.3 Retrieval of Ruby procedures' information ......... 19
   5.4 Web methods generation ............................... 20
      5.4.1 Static web methods generation ................... 20
      5.4.2 Parser identified web methods generation ....... 21
      5.4.3 Default methods generation ...................... 23
   5.5 How to use KWATT ..................................... 25
   5.6 Future work ........................................... 25
6 Automatic generation of Ruby/Tk GUIs
   6.1 KWATT GUI Tools overview ..................................... 26
   6.2 GUI Generator overview .......................................... 30
   6.3 Technologies used .................................................. 31
   6.4 Ruby Plugin Architecture ......................................... 31
      6.4.1 Generating classes with wsdl2ruby ......................... 31
      6.4.2 GUI template ................................................... 33
      6.4.3 Generated GUI ................................................ 36
   6.5 Future work .......................................................... 38

7 Conclusion ................................................................. 39

Appendices ................................................................. 42
   A Ruby script example ................................................. 42
   B Web methods generated for multipleWidgets.rb .................. 45
   C Script generated for the GUI ...................................... 58

Automatic generation of visualization web services 3


## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>KWATT services generation process from Tcl scripts</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>KWATT communication between client and server</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>KWATT Project</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>Web service generation process</td>
<td>17</td>
</tr>
<tr>
<td>5.2</td>
<td>Parsers classes diagram</td>
<td>18</td>
</tr>
<tr>
<td>6.1</td>
<td>Kwatt GUI Tools Preferences panel</td>
<td>27</td>
</tr>
<tr>
<td>6.2</td>
<td>Kwatt GUI Tools interface</td>
<td>27</td>
</tr>
<tr>
<td>6.3</td>
<td>Kwatt GUI Tools</td>
<td>29</td>
</tr>
<tr>
<td>6.4</td>
<td>Graphical User Interface generation process</td>
<td>30</td>
</tr>
<tr>
<td>6.5</td>
<td>Generated GUI for calculator.rb</td>
<td>36</td>
</tr>
<tr>
<td>6.6</td>
<td>Generated GUI for multipleWidgets.rb and Visquad.tcl</td>
<td>37</td>
</tr>
</tbody>
</table>
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>CMakeLists.txt example</td>
<td>18</td>
</tr>
<tr>
<td>5.2</td>
<td>CMakeLists.temp example</td>
<td>18</td>
</tr>
<tr>
<td>5.3</td>
<td>Method example in a Ruby script</td>
<td>19</td>
</tr>
<tr>
<td>5.4</td>
<td>Keyword example for a procedure</td>
<td>20</td>
</tr>
<tr>
<td>5.5</td>
<td>Method example in MyMethod.temp</td>
<td>21</td>
</tr>
<tr>
<td>5.6</td>
<td>Method signature in MyService.gsoap</td>
<td>21</td>
</tr>
<tr>
<td>5.7</td>
<td>VTK web method core</td>
<td>22</td>
</tr>
<tr>
<td>5.8</td>
<td>Web method core</td>
<td>22</td>
</tr>
<tr>
<td>5.9</td>
<td>Render window method</td>
<td>23</td>
</tr>
<tr>
<td>5.10</td>
<td>Static web method for the rotation</td>
<td>23</td>
</tr>
<tr>
<td>5.11</td>
<td>Web method of rotation with mouse interaction</td>
<td>24</td>
</tr>
<tr>
<td>6.1</td>
<td>GUI elements for a VTK Ruby procedure</td>
<td>28</td>
</tr>
<tr>
<td>6.2</td>
<td>Example of an XML file containing GUI informations</td>
<td>29</td>
</tr>
<tr>
<td>6.3</td>
<td>wsdl2ruby command</td>
<td>32</td>
</tr>
<tr>
<td>6.4</td>
<td>Skeleton code generated with wsdl2ruby</td>
<td>32</td>
</tr>
<tr>
<td>6.5</td>
<td>Idback method code with wsdl2ruby</td>
<td>32</td>
</tr>
<tr>
<td>6.6</td>
<td>Creation of the script calling wsdl2ruby</td>
<td>32</td>
</tr>
<tr>
<td>6.7</td>
<td>applyTemplate method of Ruby Plugin</td>
<td>33</td>
</tr>
<tr>
<td>6.8</td>
<td>Files included in the Velocity template</td>
<td>34</td>
</tr>
<tr>
<td>6.9</td>
<td>idback method in the Velocity template</td>
<td>34</td>
</tr>
<tr>
<td>6.10</td>
<td>GUI method for the rotation</td>
<td>34</td>
</tr>
<tr>
<td>6.11</td>
<td>Example of a method in the Velocity template</td>
<td>35</td>
</tr>
<tr>
<td>A.1</td>
<td>Script multipleWidgets.rb</td>
<td>42</td>
</tr>
<tr>
<td>B.1</td>
<td>Web methods generated for multipleWidgets.rb</td>
<td>45</td>
</tr>
<tr>
<td>C.1</td>
<td>Script Ruby generated for the GUI</td>
<td>58</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Nowadays, the Web is becoming a service provider as Web-accessible programs are created more and more easily each day. However, the task of developing such web services is still reserved to people with computer science skills, whereas many scientists garner a great deal of interest for these.

The aim of the School of Computational Science, which is composed of scientists coming from various backgrounds and computer scientists, is to solve engineering problems with algorithms. However, making their programs act like web services can be really tedious. In this view, a project was created for developing an automation tool for automatically generating visualization web services.

As an intern, I had the responsibility to extend their existing application with new functionalities. Moreover, in order to enable clients to communicate in the most effective way with such generated web services, a GUI generation tool was also needed. As a result, I also collaborated on developing a part of this GUI generation tool.

After a brief presentation of the working environment and the context of the project, a study of the existing applications will be made. We will then dwell on the project management before describing in details the two different parts of my work. As a conclusion, I will present an overview of my technical realizations and what this internship brought me in a professional and personal ways.
Chapter 2

Presentation

Florida State University, also known as FSU, is an American university founded in 1851 and located in Tallahassee, Florida’s capital. It provides more than 300 programs of studies in various domains such as science, cinema, engineering, art, etc. It also delivers each year more than 2000 diplomas [14].

The School of Computational Science is working on the creation of scientific computing activities, in order to solve scientific and engineering problems. Many scientists with different backgrounds, such as Physics or Mechanical Engineering, are working alongside computer scientists for creating and applying such tools [8].

This school is also a member of the NSF$^1$ funded VLab [4] consortium, which is an interdisciplinary consortium dedicated to the development and promotion of the theory of planetary materials.

As applications get more and more complex each day, the need to facilitate the development of tools for accessing and manipulating data is rapidly increasing. A remote application can be the solution of many problems, such as the data location and managing access to multiple users at the same time. Web services bring us a solution in defining a standard invocation procedure and a way to retrieve the results across the Web, in the most effective way.

In the School of Computational Science, using web services, scientists could easily access and visualize their work. However, the task of creating such visualization web services can be quite tedious. As a result, they worked on creating automation tools in order to generate visualizations web services directly from a script, starting with a VTK script in Tcl: the Web Automation and Translation Toolkit (WATT) compiler, and the Kill-a-WATT (KWATT) toolkit.

The second tool is still a work in progress, and my work was to upgrade it with new functionalities detailed in the later parts, under the instruction of Dr. Gordon Erlebacher, and in a team composed by Yenan Qu, Evan Bollig, Julien Lafourcade and Nicolas Masson.

$^1$National Science Foundation
Chapter 3

Related Work

Before giving an insight into the two toolkits of automatic generation of visualization web services already developed, it’s necessary to gain a better idea on what is exactly a web service.

3.1 Introduction to web services

Web services are a communication mechanism between remote applications on the Web, independent to any operating system or programming language. They use the HTTP protocol as a transportation means. That way, communications are made on an universal support, which is usually not filtered by firewalls. They also use an XML syntax in order to describe remote function calls and data exchanges. Finally, they organize the call and response mechanisms.

The web service mechanism is based on a three layers architecture:

- XML messaging, for describing the structure of the exchanged messages by the applications,
- Discovery, for searching and localising a particular web service in a web services registry,
- Description, which goal is to provide an interface description of the web services.

3.1.1 XML messaging

There are two standards commonly used by the web services, each one based on XML:

- XML-RPC (XML Remote Procedure Call) where the requests are sent via HTTP POST, but it doesn’t provide a service description grammar essential for some applications,
- SOAP which can use many transport protocols, XML schemas and namespaces. It enables client applications to easily connect to remote services and invoke remote methods.
3.1.2 Discovery
The most used protocol for the discovery of new web services is UDDI (Universal Description Discovery and Integration). This protocol is a specification for publishing and finding services, usually in a registry of web services.

3.1.3 Description
The WSDL standard (Web Service Description Language) is based on XML for describing web services. It describes the web service location as well as the operations (methods, parameters and return values) of the proposed service.

3.2 WATT
WATT (Web Automation and Translation Toolkit) [1] compiler was designed for converting VTK scripts written in Tcl into C++ web services. The different steps are composed by the parsing of the script, the translation in C++ and then the integration of the bindings to the gSOAP library, the C++ library for SOAP.

This compiler was written in OCaml, but only works for scripts written in Tcl. However, as this tool was too complicated and used cross-language compilation, a new toolkit called KWATT, based on the same goal of automating the generation of web services, was developed.

3.3 KWATT
KWATT (Kill-a-WATT) is a C++ application which can, given a script, generate the corresponding web service automatically in C++. Indeed, the procedures from the script will be called via generated web methods. At the development beginning, only Tcl scripts for VTK were considered but more implementations were made during the last few months.

The technologies used for this application are detailed in the next part, followed by a description of KWATT architecture and how this toolkit works. A criticism on the existing KWATT is then made in order to explain why some functionalities were needed and on what I focused my work.

3.3.1 Technologies used
gSOAP
It’s a development library for generating C++ source codes for web services and client applications. It offers an XML to C/C++ language binding to ease the development of SOAP web services in C/C++. [6]

VTK
The Visualization ToolKit [13] is an open-source library in C++ for visualizing data in 2D/3D, processing images, etc. It contains several interpreted interface layers, such as Tcl/Tk, Python and Java. It’s used by thousands of researchers
and developers around the world as it works on every Unix-based platform, PCs, and Mac OS X Jaguar or later.

Tcl

The Tool Command Language [7] is a very powerful but easy to learn dynamic programming language suitable for a wide range uses, including web and desktop applications, networking, administration, testing and more. Open-source and business friendly, Tcl is a mature yet evolving language that is truly cross-platform, easily deployed and highly extensible.

Tk

Tk [12] is a graphical library flexible, easy to learn, cross-platform and extensible. First created for Tcl, it’s interfaced with other scripting languages such as Python, Perl and Ruby. It proposes a graphical components set called widgets. Tk also contains a standard library called TkLib, written in Tcl, for visualizing data and managing the layout of the widgets.

Python

It’s an interpreted language which can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code. [9]

Cmake

It’s a cross-platform system for build automation, comparable as Unix Make program. The build process is controlled by a configuration file called CMakeLists.txt. Unlike Make, it doesn’t build the final software, but generates standard build files [3].

3.3.2 KWATT architecture

Application launch

This application takes as input a Tcl script which is parsed to retrieve information on the procedures of this script. With these information, the parser generates C++ output files containing the corresponding code for these procedures, with some static web methods (See figure 3.1).

The generated C++ files are sent to the gSOAP compiler, which generates the WSDL file and SOAP bindings for these web methods. The C++ compiler then generates executable web service.

Finally, the service is launched on a port and listens to input from clients. That means that KWATT is a stand-alone application.
Chapter 3. Related Work

Communication between server and clients

After being launched, the server waits for asynchronous SOAP messages from clients (See figure 3.2). Each time a client connects itself to the server, the server opens a socket for the client, and returns a unique id and image port number to the client via the idback() method which is published as a static web method. This method calls cconnect(), which opens the socket in order to receive later some image data if needed.

When a client calls a method that will send back image data, the server will verify the client’s unique id and their IP addresses before pushing the data through the socket. Otherwise, a message with the response is sent to the client via gSOAP.

3.3.3 Criticism

This application only worked at first for Tcl scripts, and the Python implementation was still a work in progress. In order to make it really effective for scientists who use any kind of VTK scripts, this application needs to have the ability to parse through all the existing scripting languages.

Moreover, in order to fully appreciate and use this automation tool, a GUI generation tool is also needed. The clients will not have to program themselves a way to access the generated web services.
Chapter 3. Related Work

Figure 3.2: KWATT communication between client and server
Chapter 4

Project Management

4.1 Project specifications

4.1.1 Project goal

According to the needs expressed in part 3.3.3, my work was to extend KWATT in order to create auto-generated web services from Ruby scripts, and participate in the development of a plugin for the GUI generation tool that Julien Lafourcade developed during his internship (See figure 4.1).

4.1.2 Specifications

My work was divided in different steps:

- a study of the existing application KWATT,
- the creation of a parser for Ruby scripts in this application,
- the development of new functionalities such as generating non-VTK web services, or giving the ability to generate one web service for multiple scripts as input,
- the development of a Java plugin for generating Ruby/Tk GUI for these web services,
- the development of the mouse interaction in the plugin and KWATT.

4.1.3 Working environment

Throughout my internship, I worked on a Fedora system, using Emacs for the development in C++ for KWATT, and Eclipse IDE for the plugin development. The technologies used for each part of my work are described in the next chapters.
Chapter 4. Project Management

Figure 4.1: KWATT Project

Automatic generation of visualization web services
4.2 Realization

Throughout the development, we worked on communicating regarding the march of our respective works. Every three weeks, we organized a meeting for presenting our work using slides, videos, live tests, etc. New ideas concerning the development were exchanged, and solutions for possible problems were given.

Every major step of the development was tested on different computers and for KWATT, the code was updated every time using SVN (SubVersion repository).

4.3 Documentation

In order to simplify the future development, a documentation is needed for each part. As for my work on KWATT, a documentation was made using Doxygen, and is available on the SVN repository.

At this time, the documentation for the plugin is still a work in progress.
Chapter 5

Automatic services generation from Ruby scripts

After studying in details the existing application, I focused my work on the auto-generation of web services from Ruby scripts.

Before explaining in details how this implementation is done, the technologies used are described in the next section. We will then see the different modifications that I had to make on KWATT architecture, followed by details on the main steps of the web service generation. A reflexion on the future of KWATT ends this chapter.

5.1 Technologies used

These technologies complete the list of those used in KWATT, as previously seen in the part 3.3.1.

Ruby

It's an interpreted language, cross-platform and object-oriented. Besides being very intuitive, Ruby can be used as a scripting language in the same way than Perl or Python. [10]

VTK-Ruby

It’s a patch for VTK, in order to use it from Ruby. That way, VTK also has an interface layer for Ruby. [16]

5.2 Modifications to KWATT architecture

Whereas the web service generation process is the same than the one described in the 3.3.2 part, as shown in the figure 5.1, the core of the application differs
Chapter 5. Automatic services generation from Ruby scripts

5.2.1 Object-Oriented KWATT

In the first version of KWATT, the Tcl implementation was the only efficient one, the Python one was still a work in progress. With the handling of new types of input scripts, new parsers need to be implemented. Indeed, as the way of defining procedures differs from one language to another, the retrieval of these procedures’ information is quite different. However, some methods used by the parsers are commons, like for instance the ones for managing the template files needed for the generation. As a result, a superclass Parser defines all those methods commonly used.

With the implementation of the Ruby parsing, some new functionalities were also developed. One of them was to enable KWATT to generate non-visualization web services, meaning that the input script can be a non-VTK one. As the generation between these types of web services is different, even if the script language is the same, the solution was to create one class for each types of generated web services (See figure 5.2).

With this object-oriented implementation, the addition of new parsers is simplified. In the future, developers will just have to extend this superclass and create their own parser implementation.
5.2.2 CMake configuration file generation

As we saw previously, *cmake* is a tool for build automation controlled by a configuration file called *CMakeLists.txt*. As the generated files need different header files and libraries, the *CMakeLists* needs to be also generated according to the script’s type. For example, if the input script is written in Ruby, in order to correctly compile, the *CMakeLists* needs to specify the path to Ruby header files and libraries. However, if the input script is in Tcl or Python, it doesn’t have to specify these Ruby paths, but only the ones needed for Tcl or Python. Here’s an example on what can be found in a *CMakeLists* for a Ruby script :

```
... INCLUDE_DIRECTORIES ( 
  ... $ENV[RUBY_INCLUDE_PATH] 
) LINK_DIRECTORIES ( 
  ... $ENV[RUBY_LIBRARY] 
) LINK_LIBRARIES( 
  ... ruby 
) ... 
```

The paths to the header and libraries are given by the environment variables previously created.

As this configuration file depends on the type of input scripts, each parser needs to fill this file before the parsing. A template *CMakeLists.temp* is designed for this task: with special keywords, it shows where to put the corresponding information.
Chapter 5. Automatic services generation from Ruby scripts

Listing 5.2: CMakeLists.temp example

```c
 INCLUDE_DIRECTORIES ( 
    #@@VTK_INCLUDE@@
    #@@RUBY_INCLUDE@@
 )
 LINK_DIRECTORIES ( 
    #@@VTK_LINK@@
    #@@RUBY_LINK@@
 )
 LINK_LIBRARIES ( 
    #@@VTK_LINKLIB@@
    #@@RUBY_LINKLIB@@
 )
```

As we want to be able to generate one web service from multiple scripts, this template configuration file needs to be opened only in the beginning of the application, filled with new information when each script is parsed, and copied into the `CMakeLists.txt` at the end of the parsing. That way, if a Tcl and a Ruby scripts are given, the `CMakeLists.txt` will contain the header files and libraries for both languages.

### 5.3 Retrieval of Ruby procedures’ information

After generating the configuration file for `cmake`, the input script needs to be parsed. As there are now different kind of parsers, the file extension and the option given when launching the application tells KWATT which one to apply. For specifying if it’s a VTK script, the option ”-vtk” is given before the script’s name. The chosen parser will skim through the file in order to get the methods’ name and their parameters.

To save these data, we need to define some structures. For the Ruby implementation, these structures are defined in the RubyParser header file. It defines a structure to store a Ruby procedure’s information, with the name, a return type and a list of parameters. The name of the class where the method was defined can also be specified if there’s any.

If the input script is a VTK one, the name of the render window can also be retrieved. A render window in VTK is an object which renders the 3D scene described in the script. The name can be found in the script next to the keyword `KW_RENDER_WINDOW`, which specifies that the method modifies the state of an object in the VTK scene.

Listing 5.3: Method example in a Ruby script

```
Automatic generation of visualization web services
19```
@setColor = Proc.new{|i, j, k| #KW_RENDER_WINDOW renWin
  nb = 0
  while nb < 4 do
    actors[nb].GetProperty.SetColor(i, j, k)
    nb = nb + 1
  end
}

In this example, for the procedure "setColor", the render window name is "renWin", and there are three parameters.

An argument structure is also defined, with a name and a type. Ruby is a scripting language based on the principle of duck-typing: an object's type is determined by what it can do, not by its class. In that case, the parser doesn’t know how to find the types. There’s two possibilities:

- by default, we specify the type of an argument to be the one for strings in gSOAP, `xsd_string`,
- or we specify the types directly in the input script using a new keyword.

In the last case, the new keyword `KW_KEY` is used for declaring a procedure and its parameters' type. For example, for an `add` procedure which takes two float and returns a float, in a Ruby calculator script, we have to find the following line in the script:

```
#KW_KEY add float float float
```

If the procedures in the original script are defined after the keyword `KW_PUBLIC`, the structures are filled during the first step of the parsing. Otherwise, the parser considers the methods to be private, and will not collect data from these.

At the end of this step, a `RubyVtkParser` object has a list of procedures and a list of render window objects, whereas a `RubyDefaultParser` only has a list of procedures.

### 5.4 Web methods generation

As seen in figure 5.1, the generation is divided in two: the generation of static web methods and the generation of parser identified web methods. However, some default web methods were also added for a visualization web service, such as applying a rotation, translation and zoom on the scene.

#### 5.4.1 Static web methods generation

The static web methods are the one generated for each web service by default. They are defined in a template file `MyMethod.temp`. This file will be filled with the definitions of the new web methods, as explained in the next part.
Chapter 5. Automatic services generation from Ruby scripts

In these static web methods, we can find the methods needed for managing clients connections. If KWATT service is launched without any input script, we can also find a new web method for generating a new KWATT service with a new script. That way, clients can submit their own script for generating the corresponding web services. The file is uploaded on the server, and after the generation, the new KWATT service is launched.

5.4.2 Parser identified web methods generation

When the parsing of the script is done and the structures are filled, the corresponding web methods are generated into two template files: `MyMethod.temp` and `MyService.gsoap`. The first file contains the web methods definitions, the second one contains the web methods declarations for gSOAP. For example, if in the Ruby script there’s a procedure called `setColor` which takes three arguments, the signature of the generated web method in `MyMethod.temp` will be:

```
Listing 5.5: Method example in MyMethod.temp

    int ns__rbMultipleWidgetsSetColor(SOAP* soap, xsd__string r, xsd__string g, xsd__string b, int cid, ns__rbMultipleWidgetsSetColorResponse* result);
```

And the following lines will be generated in `MyService.gsoap`:

```
Listing 5.6: Method signature in MyService.gsoap

    class ns__rbMultipleWidgetsSetColorResponse {
        xsd__string response;
    };

    int ns__rbMultipleWidgetsSetColor(xsd__string r, xsd__string g, xsd__string b, int cid, ns__rbMultipleWidgetsSetColorResponse* result);
```

The name of the web method is prefixed with the type and the name of the script. It’s for avoiding problems when multiple scripts are loaded, where some methods can have the same name. For example, if in a Tcl script and in a Ruby script there is a method called `setColor` with the same number of parameters, there would be a compilation error. The best way to prevent this is to give them a different name.

The web methods generated need to call the corresponding method of the input script. For the Ruby implementation, I needed to find a way to call the Ruby methods within the C++ code of the web service. The solution is to use Ruby embedded functions in C [5]. It permits to call the Ruby interpreter within the application and access the methods from the Ruby script that is loaded. For calling the Ruby embedded functions, the Ruby header files, the Ruby static library and the Ruby dynamic link library are needed. To run a Ruby expression in C code, we use the method `rb_eval_string(const char*)` [17]. But first, the Ruby interpreter needs to be initialized.

*Automatic generation of visualization web services*
Chapter 5. Automatic services generation from Ruby scripts

In order to initialize only once this interpreter, a class called `RubyInterpreter` was created, containing a method `ensureRubyInitialized()`. This method verifies if the Ruby interpreter was previously launched or not. If it wasn’t, the methods `ruby_init()` and `ruby_init_loadpath()` are called. The Ruby load path is initialized with the content of the RUBYLIB environment variable, which is searched whenever a Ruby script is required.

After being sure that the Ruby interpreter was previously launched, the Ruby script needs to be loaded via the embedded method `rb_require(const char*)`, at the beginning of the service. That way, this file is accessible whenever a web method is called.

Now that the script has been loaded and the parser identified web methods can be generated. However, the generation of these web methods is different if the input script is a VTK one or not.

If the procedure comes from a VTK script and updates the state of a render window, we need to render the result and retrieve the image data. This is done with the creation of a pointer on the result of the call `rb_eval_string()` with the name of the render window. The scene is rendered on the server side, and the image data is written directly to memory. Finally, this data is sent to the client through the socket.

In the following listing, there’s an example of a call to the method `setColor`, with the creation of the pointer to the render window object.

Listing 5.7: VTK web method core

```
// method call
rb_eval_string("setColor.call(0,1,1)");
// pointer on the render window object
VALUE ptr = rb_eval_string("@renWin");
rBVTKSpecialObject *obj;
Data_Get_Struct(ptr, rBVTKSpecialObject, obj);
void *ptr2 = obj->ptr;
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin = (vtkRenderWindow*)ptr2;
```

This particular development was a little hard to realize because of the absence of documentation on the embedded functions in C of Ruby/Vtk. These embedded functions are available when installing the package, but no indications on how to use them are given, contrarily to the Ruby embedded functions. After some tests, I finally found a way to retrieve a pointer on a Ruby/VTK object.

For a procedure where a result is needed, we just retrieve the result and send it directly to the client via gSOAP.

Listing 5.8: Web method core

```
// method call
```
Chapter 5. Automatic services generation from Ruby scripts

5.4.3 Default methods generation

Besides these web methods corresponding to the procedures found in the script, and if it’s a visualization web service that is generated, a method is defined for each render window object in order to send the image data to the client. It gives the ability to visualize the scene without having to do modifications via the other web methods.

Their names are composed by the type and the name of the script, the name of the render window and the suffix "back". The source code is based on the template file `rb_img_web_methods.temp`, and works the same way than the code from the VTK web method for rendering the scene.

Listing 5.9: Render window method

```ruby
int ns_rmbmultipleWidgetsrenWinback(SOAP* soap, int cid,
  ns_rmbmultipleWidgetsrenWinbackResponse* result) { ...
  // pointer on the render window object
  VALUE ptr = rb_eval_string("@renWin");
  rbVTKSpecialObject *obj;
  Data_Get_Struct(ptr, rbVTKSpecialObject, obj);
  void *ptr2 = obj->ptr;
  vtkRenderWindow *renWin = vtkRenderWindow::New();
  renWin = (vtkRenderWindow*)ptr2;
  ...
}
```

One of the new ideas exchanged during our monthly presentation, was to give the ability to the client to interact with the scene in rotating, translating and zooming it, without having these methods defined in the original script.

This implementation is still a work in progress as it’s only implemented in the `RubyVTKParser` and needs to be improved. The idea was to generate these default methods as web methods during the parsing of the script, and define three static methods.

These static methods will contain the source code for each modification to the scene in C++. Each one takes a `vtkRenderWindow` and the parameters needed for applying the transformation.

Listing 5.10: Static web method for the rotation

```ruby
void rotate(vtkRenderWindow *renWin, double angle){
  vtkRendererCollection *renderers = renWin->GetRenderers();
}
```
Chapter 5. Automatic services generation from Ruby scripts

3

vtkRenderer *ren = renderers->GetFirstRenderer();
vtkTransform *trans = vtkTransform::New();
trans->RotateWXYZ(angle, -1, -1, -1);
ren->GetActiveCamera()->ApplyTransform(trans);
}

These methods are called in the web methods generated for each render window object during the parsing. There are two web methods for the rotation, two for the translation, and one for the zoom. In fact, a rotation can be made according to two different ways: in a graphical interface, where the mouse interaction is taken into account and the angle is calculated with the mouse coordinates, or in a command, where the client specifies directly a rotation angle. This reason also applies to the translation. The implementation of the mouse interaction for the graphical user interface will be explained in the next chapter.

For each one of these default web methods, a template file is used for generating the source code. The name of the method is composed by the type and name of script, like the other web methods generated, and by the name of the render window object.

If the method needs calculation for the parameters for the transformation, this is done first in these web methods. Then, just like the method for rendering the scene, a pointer on the render window object is created. Finally, the static web method is called with the render window object created and the parameters needed. Here’s an example of the web method for the rotation which can be called in a graphical user interface with mouse interaction:

Listing 5.11: Web method of rotation with mouse interaction

```ruby
int ns_rbmultipleWidgetsrotationGUI(SOAP* soap, double sizeImgX, double sizeImgY, double posFirstX, double posFirstY, double posLastX, double posLastY, xsd_string renwinName, int cid, ns_rbmultipleWidgetsrotationGUISResponse* result) {
...
// rotation angle calculation
...
double angle = startangle - stopangle;
// retrieval of a pointer on the render window object
string rw = "@" + renwinName;
VALUE ptr = rb_eval_string(rw.c_str());
// creation of a vtkRenderWindow object
rbVTKSpecialObject *obj;
Data_Get_Struct(ptr, rbVTKSpecialObject, obj);
void *ptr2 = obj->ptr;
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin = (vtkRenderWindow*)ptr2;
// call the rotate method with the angle and the render window object
rotate(renWin, angle);
```
When all the scripts given have been parsed, all the files needed for the web service are copied into the current directory via the \textit{Parser} method \texttt{write-FromTemplate(string, string)}. The generated web service is then launched.

A Ruby script example can be found at the appendix A.1. The web methods generated from this example can also be found at the appendix B.1.

5.5 \textbf{How to use KWATT}

There are three ways to launch the KWATT application now:

- with the \texttt{vtk} option, meaning that the script contains VTK code ("kwatt -vtk script"). In that case, KWATT will generate a visualization web service from the input script.

- with just the name of a script, meaning that it’s not a VTK script ("kwatt script"). In that case, KWATT will generate a default web service from the input script.

- without any arguments ("kwatt"). In that case, KWATT will generate a web service containing only the static methods. Clients can send their own script for generating a new KWATT web service.

5.6 \textbf{Future work}

Regarding the implementation that I made on KWATT, it can be useful to simplify the code with using an existing tool that can parse through files and collect data. A tool for creating template files can also improved the development. By lack of time, such solutions were not exploited during this internship, but everything was made with keeping this consideration at mind.

In the future, an implementation of Perl in KWATT will be necessary in order to cover a wide range of scripts. It could also be possible to make this application work for any kind of program, written in Java, or any language, based on the same principle.

It would be also interesting to give KWATT the ability to parse through scripts for another visualization tool, such as Amira, Matlab, etc. That way, scientists could work with KWATT and still be able to visualize their work without having it to be restricted to VTK.
Chapter 6

Automatic generation of Ruby/Tk GUIs

Using KWATT application, web services can be generated from scripts written in Tcl, Python and Ruby. However, these web services can only be used from client applications, and more likely, through a Graphical User Interface (GUI). The generation of these web services creates complications in the user interaction process. It is therefore important to develop approaches to automatically generate the graphical user interfaces.

In this view, Julien Lafourcade developed a GUI Generator, acting as a web service, and a client application called Kwatt GUI Tools for communicating with this web service in order to generate a GUI from KWATT services.

As the GUI Generator is based on a plugin architecture for enabling to generate GUIs in multiple languages, my work was to develop a plugin written in Java in order to generate Ruby/Tk GUIs.

An overview on the Kwatt GUI Tools and the GUI Generator will be first presented, followed by an explanation in details of the plugin developed.

6.1 KWATT GUI Tools overview

Kwatt GUI Tools is a client application that enables clients to obtain a GUI for the generated KWATT web services.

The first step of this application is to specify in the Preferences panel, the configuration for KWATT and the Generator. Clients need to specify the URL and the port of connection of these services (See figure 6.1).

The next step is to provide this application a script from which a GUI can be generated (See 1 in figure 6.2). That means that we have to find a way to specify what GUI elements could be created. The chosen solution is to define keywords directly in the script.
Chapter 6. Automatic generation of Ruby/Tk GUIs

Figure 6.1: Kwatt GUI Tools Preferences panel

Figure 6.2: Kwatt GUI Tools interface

Automatic generation of visualization web services
Chapter 6. Automatic generation of Ruby/Tk GUIs

Listing 6.1: GUI elements for a VTK Ruby procedure

```ruby
@setColor = Proc.new{|i, j, k| #CRENDERWINDOW renWin
#UI IMAGE
#UI ELEMENT tf1 textField
#UI ELEMENT tf2 textField
#UI ELEMENT tf3 textField
#UI PARAM tf1
#UI PARAM tf2
#UI PARAM tf3
  nb = 0
  while nb < 4 do
    actors[nb].GetProperty.SetColor(i, j, k)
    nb = nb + 1
  end
}
```

In this example, we can see that the keywords for the GUI generation are prefixed with "#UI". The keyword ELEMENT specifies that there are three GUI elements, which are textfields, and IMAGE specifies that this method modifies a render window object.

In KWATT, as we saw previously, there’s a static method for enabling clients to start their own KWATT service with a new script. In Kwatt GUI Tools, this functionality is provided with the possibility to start a new KWATT service (See 2 in figure 6.2 and 1 in figure 6.3). The Kwatt Deployer, a component of this application, sends this script to KWATT using the static web method new_service(string file). The port of the new service is then retrieved for creating the WSDL URL used by the Generator Client.

If clients choose this option, they don’t need to specify the service URL and the port of connection (see 3 in figure 6.2). Otherwise, they have to specify them. The various plugins availables are also given.

When clients click on “Start”, the application first create a new KWATT service if the option was chosen, then parses the given script for retrieving GUI information using the GUI parser (see 2 in figure 6.3). The parser looks for the GUI elements keywords and fills a data structure for the Generator Client.
When all data have been collected, with the GUI information and the WSDL URL, an XML file is generated. As an example, here’s what such XML file looks like:

**Listing 6.2: Example of an XML file containing GUI information**

```
<procedure name="rbmultipleWidgetssetColor">
  <parameters>
    <parameter>
      <name>i</name>
      <guiElementId>tfl</guiElementId>
      <rangeFormat>noRange</rangeFormat>
    </parameter>
    ...
  </parameters>
  <guiElements>
    <guiElement>
      <id>tfl</id>
      <type>textfield</type>
      <options></options>
    </guiElement>
    ...
  </guiElements>
  <presets>
    <renderWindowName>renWin</renderWindowName>
    <renderWindowNameWithPrefix>rbmultipleWidgetsrenWin</renderWindowNameWithPrefix>
  </presets>
</procedure>
```
renderWindowNameWithPrefix

Then, the Generator is called via the Generator Client (See 3 in figure 6.3). An archive containing the files generated is sent back by the Generator, which is created locally by the Generator Client at the end of the GUI generation (See 4 in figure 6.3).

6.2 GUI Generator overview

The generated XML file from the Generator Client is parsed for creating a model containing all the GUI informations (See 1 in figure 6.4).

With this GUI information model, a GUI is generated, using one of the plugins available (See 2 in figure 6.4). At first, there was only a Java plugin for generating a GUI in Java/Swing. The files generated are finally collected in an archive which is sent to the client.

My work was to develop a new plugin written in Java to automatically generate a Ruby/Tk graphical interface from a script written in Tcl, Python or Ruby.
6.3 Technologies used

Axis2

Axis is a project of the Apache Software Foundation. It’s a free Java package which provides an environment that can serve as a SOAP server, an API for web services development, various transport layers, tools for automatically generate the WSDL files from Java classes or vice-versa, and some tools used for deploying, testing and managing web services. Axis2 [2] is the complete re-write of Axis in a much efficient way, more XML oriented.

wsdl2ruby

It’s a Ruby script from the soap4r [11] package, that can generate Ruby class definitions from a WSDL file.

Velocity

Java-based template engine developed by the Apache Software Foundation, a simple and powerful development tool that allow us to easily create and render documents that format and present our data. For example, it permits web designer to reference methods defined in Java code. Velocity separates Java code from the web pages, making the web site more maintainable over the long run. [15]

6.4 Ruby Plugin Architecture

In the Ruby Plugin, a class called RubyPlugin implements the IPlugin interface provided by the Generator. It contains a method generateGui(), which takes a IGui object and a temporary directory where the files will be generated. The first step of this method is to generate the files needed for the communication with the web service. The second step is the generation of the Ruby file that defines the GUI for this web service.

6.4.1 Generating classes with wsdl2ruby

To create applications capable of communicating with web services via SOAP, a tool that can automatically generate class definitions for creating objects, mapping those to SOAP and vice-versa, is needed.

This is exactly what soap4r and the wsdl2ruby script provide. soap4r is an additional package that provides tools for using SOAP within Ruby code. Using soap4r, a developer can easily generate both client and server classes to handle consuming and providing SOAP-accessible services. For developing a GUI that can communicate with web services, the generation of client code is only needed.

The wsdl2ruby script does exactly what its name implies : it takes a WSDL definition of a web service and transforms it in different class definitions in Ruby.
Chapter 6. Automatic generation of Ruby/Tk GUIs

code.

The command for generating client code for a web service from a WSDL definition with wsdl2ruby is:

Listing 6.3: wsdl2ruby command

```
wsdl2ruby.rb --wsdl wsdlUrl --type client
```

This command will generate three files:

- **MyServiceClient.rb**: an example file that provides skeleton code for communicating with the web service,
- **MyService.rb**: the set of class definitions for all elements defined by the WSDL file. Using these class definitions, a developer will be able to interact with the web service,
- **MyServiceDriver.rb**: this file contains a single class *MyServicePortType*, which is used to conduct all requests to the web service.

*MyServiceClient.rb* provides a skeleton application that initializes *MyServicePortType* object, and then uses it to call each of the operations made available by the web service. The *wsdl2ruby* application auto-generates a skeleton for each operation that looks like this:

Listing 6.4: Skeleton code generated with wsdl2ruby

```
parameters = nil
puts obj.idback(parameters)
```

The parameters are set to nil, whereas the web service requires an *Idback* object. The parameters required to create these objects are determined by the WSDL definition, and the order of parameters to pass to *new* are documented in part in *MyService.rb*. For example, for the *idback* method we obtain the following code:

Listing 6.5: Idback method code with wsdl2ruby

```
parameters = Idback.new
idbackResponse = obj.idback(parameters)
```

The implementation of this in the RubyPlugin will be explained in the GUI template part.

In order to call *wsdl2ruby* from the RubyPlugin, a script called *make.sh* is created during the first step of the generation.

Listing 6.6: Creation of the script calling wsdl2ruby

```
fosLaunch = new FileOutputStream(temporaryDirectory + "\make.sh", false);
```
Chapter 6. Automatic generation of Ruby/Tk GUIs

6.4.2 GUI template

The second step in the method `generateGui()` of this plugin is the generation of a Ruby/Tk [18] script from a Velocity template that will define the GUI for the web service. It’s made with the creation of a `TemplateFiller` object. This object has two methods: `init()` and `applyTemplate()`.

The `init()` method initializes Velocity with `Velocity.init(p)`, where `p` is a set of properties, which can contain the jar files needed. The `applyTemplate()` method is then called with a GUI model, a template file and an output directory as parameters. A Velocity context is defined with the GUI model, in order to access the GUI information within the template in a more effective and simplified way, and the Ruby file is generated with the template specified.

Listing 6.7: `applyTemplate` method of Ruby Plugin

```
public String applyTemplate(IGui model, String templateFile, String outputDir) throws Exception {

    // creation of the Velocity context
    VelocityContext context = new VelocityContext();

    // informations added to the context
    context.put("className", className);

    // apply the template
    Template template = Velocity.getTemplate(templateFile);

    DataOutputStream dosLaunch = new DataOutputStream(fosLaunch);
    dosLaunch.writeBytes("#!/bin/sh
    
    ruby " + model.getGuiName() +"GUI.rb
    
    
    
    BufferedReader writer = new BufferedReader(new FileWriter(outputDir + "/" + model.getGuiName() +"GUI.rb")

    In the Velocity template file, the first step is to include the Ruby files needed, such as `tk` for the graphical interface, `tkextlib/tkimg` for displaying a png file, `socket` for the connection and the data retrieval to the socket, and the `MySer-
viceDriver class that was generated with wsdl2ruby.

Listing 6.8: Files included in the Velocity template
```ruby
require 'tk'
require 'MyServiceDriver.rb'
require 'tkextlib/tkimg'
require 'socket'
```

A class named after the web service is then created, where will be defined the methods. Some default methods first need to be defined, then the methods for accessing the web methods, for finally generating a method that will create the interface.

**Default methods**

The most important default method that needs to be defined is idback that calls the web service’s method at the connection in order to retrieve the client id and port. The connection to the socket is also realized.

Listing 6.9: idback method in the Velocity template
```ruby
def idback
  parameters = Idback.new
  idbackResponse = @obj.idback(parameters)
  @cid = idbackResponse.id_response
  @port = idbackResponse.port_response
  @sock = TCPSocket.new("127.0.1.1", @port)
end
```

The other default methods are the ones needed for implementing the mouse interaction. For each render window objects, three methods are generated: rotation, translation and zoom. With a graphical interface, the mouse interaction can be used so the methods called will be the ones with the mouse informations on KWATT side. For example, for the rotation, the size of the image, the position of the mouse when the button is pressed, the position of the mouse when the button is released and the render window name is given as parameters to the web method.

Listing 6.10: GUI method for the rotation
```ruby
def rotation
  ...
  parameters = RbmultipleWidgetsrotationGUI.new(@sizex, @sizey, @startx, @starty, @stopx, @stopy, @renwin, @cid)
  resultRotation = @obj.rbmultipleWidgetsrotationGUI(parameters)
  ...
end
```
Web methods

In the GUI model, a web method is linked to a *guiElementGroup*. For each *guiElementGroup*, a method is created. For example, for an *add* method from a calculator script, *add* is a *guiElementGroup* and its elements are constituted of two textfields. A button is also created for each method.

When clicking on the button, the web method is called via gSOAP. If the method modify a render window object, the image data is collected via the socket from the server, and the new image is displayed in the interface.

Listing 6.11: Example of a method in the Velocity template

```ruby
1 def rbmultipleWidgetssetColor
2   parameters = RbmultipleWidgetssetColor.new(  
3     @nbrbmultipleWidgetssetColor0.value,  
4     @nbrbmultipleWidgetssetColor1.value,  
5     @nbrbmultipleWidgetssetColor2.value, @cid)
6   resultRbmultipleWidgetssetColor = @obj.
7   rbmultipleWidgetssetColor(parameters)  
8   parametersImg = RbmultipleWidgetsrenWinback.new(@cid)
9   ...  
10  threads = []  
11  threads << Thread.new(0) do  
12    while true  
13      data = @sock.recvfrom(1)  
14      ...  
15      @buf << data  
16      ...  
17    end  
18  end  
19  threads << Thread.new(1) do  
20    @resultRbmultipleWidgetsrenWin = @obj.
21    rbmultipleWidgetsrenWinback(parametersImg)  
22  end  
23  threads.each{|thr| thr.join}  
24  ...  
25 end
```

GUI definition

Finally, a method *initialize*, where all the GUI elements are created, is defined. In it, the *idback* method is called first for the connection to the web service, and the GUI elements are generated. By default, a text box is also used as a log console in the interface for displaying the web service’s responses.

If there’s a VTK scene to render, on the canvas created for displaying the image, the mouse interaction is enabled. If the user click with the left button of the mouse, the translation method is called with the new position. If the user, press the right button and releases it, the rotation method is called. Finally,
the zoom method is called when the mouse wheel is used.

An Ruby/Tk script generated using the RubyPlugin can be found at the appendix C.1.

### 6.4.3 Generated GUI

When the RubyPlugin is applied, the script generated describes a GUI in Ruby/Tk.

The generated GUI for a web service with non-VTK methods contains a menu bar with an option "Quit", a panel with all the GUI elements from each web methods, and a log console for displaying the web service’s results. The figure 6.5 shows an example of such interface.

For a web service with VTK methods, this interface will also contain a menu option "Save as ..." for saving the image currently displayed in the interface, and a canvas where the image is displayed. The figure 6.6 presents an example of a GUI generation using two scripts as input.
Figure 6.6: Generated GUI for multipleWidgets.rb and Visquad.tcl
6.5 Future work

One of the thing needed for the future, is to finalize the mouse interaction in order to simplify it and make it really effective. Some tests need to be done under various systems and using different mouses control.

The development of other plugins can also be done, in order to give clients the ability to choose the graphical interface they prefer.
Chapter 7

Conclusion

From a technical perspective, everything on the specifications was successfully realized. Thereby, KWATT now can parse through Ruby scripts in order to generate automatically web services, and with the use of Kwatt GUI Tools, Ruby/Tk graphical interfaces can now be auto-generated. Furthermore, new functionalities were also added to each tool, improving the efficiency of this project.

The work done throughout this internship can be considered as really important for the march of the project and was realized under consideration of the future developers working on it.

From a personal perspective, the developments realized gave me the opportunity to build on and increase my knowledge on programming in an object-oriented way with C++ and Java. This internship also permitted me to gain some experience on web services and to learn a new programming language, Ruby.

Furthermore, as I’ve been part of a development team, I’ve gained some experience in collaborating with other people on a common project, where everyone share and increase their knowledge.

Finally, this internship gave me the opportunity to increase my language skill in going to the United States, and participating in a project really innovative within an international team.
References


Appendices
Appendix A

Ruby script example

Listing A.1: Script multipleWidgets.rb

```ruby
##RUBY

require 'vtk'
require 'vtk/util'

cylinder = Vtk::CylinderSource.new
  cylinder.SetResolution(8)
cone = Vtk::ConeSource.new
  cone.SetResolution(8)
cube = Vtk::CubeSource.new
  cube.SetXLength(1.0)
  cube.SetYLength(1.0)
  cube.SetZLength(1.0)
sphere = Vtk::SphereSource.new
  sphere.SetRadius(1.0)

cylinderMapper = Vtk::PolyDataMapper.new
  cylinderMapper.SetInputConnection(cylinder.GetOutputPort)
coneMapper = Vtk::PolyDataMapper.new
  coneMapper.SetInputConnection(cone.GetOutputPort)
cubeMapper = Vtk::PolyDataMapper.new
  cubeMapper.SetInputConnection(cube.GetOutputPort)
sphereMapper = Vtk::PolyDataMapper.new
  sphereMapper.SetInputConnection(sphere.GetOutputPort)
cylinderActor = Vtk::Actor.new
  cylinderActor.SetMapper(cylinderMapper)
```

42
cylinderActor.GetProperty().SetColor(Vtk::Colors::Beige)
cylinderActor.RotateX(30.0)
cylinderActor.RotateY(-45.0)
coneActor = Vtk::Actor.new
coneActor.SetMapper(coneMapper)
coneActor.RotateX(30.0)
coneActor.RotateY(-45.0)
cubeActor = Vtk::Actor.new
cubeActor.SetMapper(cubeMapper)
cubeActor.RotateX(30.0)
cubeActor.RotateY(-45.0)
sphereActor = Vtk::Actor.new
sphereActor.SetMapper(sphereMapper)
sphereActor.RotateX(30.0)
sphereActor.RotateY(-45.0)
actors = [cylinderActor, coneActor, cubeActor, sphereActor]
obj = 0
ren = Vtk::Renderer.new
@renWin = Vtk::RenderWindow.new
@renWin.AddRenderer(ren)
iren = Vtk::RenderWindowInteractor.new
iren.SetRenderWindow(@renWin)
ren.AddActor(actors[obj])
ren.SetBackground(0.1, 0.2, 0.4)
@renWin.SetSize(600, 600)

# KW_PUBLIC
@setColor = Proc.new{|i, j, k| # KW_RENDER_WINDOW renWin
# UI IMAGE 2 3 4 5
# UI ELEMENT tf1 textfield
# UI ELEMENT tf2 textfield
# UI ELEMENT tf3 textfield
# UI PARAM tf1
# UI PARAM tf2
# UI PARAM tf3
  nb = 0
  while nb < 4 do
    actors[nb].GetProperty().SetColor(i, j, k)
    nb = nb + 1
  end
}
```ruby
@setRepresentationMode = Proc.new{|mode| #
  KW_RENDER_WINDOW renWin
#UI IMAGE 2 3 4 5
#UI ELEMENT tf4 choice "Wireframe" "Surface"
#UI PARAM tf4
  nb = 0
  while nb < 4 do
    if mode == "Wireframe" then
      actors[nb].GetProperty.SetRepresentationToWireframe()
    elsif mode == "Surface" then
      actors[nb].GetProperty.SetRepresentationToSurface()
    end
    nb = nb + 1
  end
}

@changeObject = Proc.new{|i| #KW_RENDER_WINDOW renWin
#UI IMAGE 2 3 4 5
#UI ELEMENT tf5 list "Cylinder" "Cone" "Cube" "Sphere"
#UI PARAM tf5
  ren.RemoveActor(actors[obj])
  ren.AddActor(actors[i])
  obj = i
}

##EOF
```

Automatic generation of visualization web services
Appendix B

Web methods generated for multipleWidgets.rb

Listing B.1: Web methods generated for multipleWidgets.rb

```c
int ns_idback(SOAP* soap, ns_idbackResponse* result) {
    if (firstcall == true) {
        pthread_t thread[2];
        pthread_create(&thread[0], NULL, test_disconnect, NULL);
        pthread_create(&thread[1], NULL, cconnect, NULL);
        rbinterp->ensureRubyInitialized();
        rb_require("multipleWidgets.rb");
        firstcall = false;
    }
    sleep(2);
    result->id_response = uniqid;
    result->port_response = listenPort;
    return SOAP_OK;
}

void rotate(vtkRenderWindow *renWin, double angle) {
    vtkRendererCollection *renderers = renWin->GetRenderers();
    vtkRenderer *ren = renderers->GetFirstRenderer();
    vtkTransform *trans = vtkTransform::New();
    cout << "angle de rotation :");
    trans->RotateWXYZ(angle, -1, -1, -1);
    ren->GetActiveCamera()->ApplyTransform(trans);
}

void translate(vtkRenderWindow *renWin, double x, double y) {
...
```c
vtkRendererCollection *renderers = renWin->GetRenderers();
vtkRenderer *ren = renderers->GetFirstRenderer();
vtkTransform *trans = vtkTransform::New();
trans->Translate(-x/500,-y/500,0);
ren->GetActiveCamera()->ApplyTransform trans; }

void zoom(vtkRenderWindow *renWin, double z)
{
  vtkRendererCollection *renderers = renWin->GetRenderers();
  vtkRenderer *ren = renderers->GetFirstRenderer();
  ren->GetActiveCamera()->Zoom(z);
}

int ns__rbmultipleWidgetssetColor (SOAP* soap,
  xsd__string i, xsd__string j, xsd__string k, int cid,
  ns__rbmultipleWidgetssetColorResponse* result)
{
  rbinterp->ensureRubyInitialized();
  string proname = "@setColor";
  string args = "("
  VALUE val[4];
  rb_eval_string("str = /[a-zA-Z]/");
  t[1].append("\n" );
  t[1].append(i);
  t[1].append("\n =\" str\" ");
  val[1] = rb_eval_string(t[1].c_str());
  if(TYPE(val[1]) == T_FIXNUM)
    { args.append("\n" );
      args.append(i);
      args.append("\n") ;
    }
  else
    { args.append("",
      rb_eval_string("str = /[a-zA-Z]/" );
      t[2].append("\n" );
      t[2].append(j);
      t[2].append("\n =\" str\" ");
      val[2] = rb_eval_string(t[2].c_str());
      if(TYPE(val[2]) == T_FIXNUM)
        { args.append("\n" );
          args.append(j);
          args.append("\n") ;
        }
    }
```
Chapter B. Web methods generated for multipleWidgets.rb

```ruby
    args.append(j);
}
    args.append("", "");
    rb_eval_string("str = /[a-zA-Z]="/);
    t[3].append("\"");
    t[3].append(k);
    t[3].append("\" =\" str\";
    val[3] = rb_eval_string(t[3].c_str());
    if (TYPE(val[3]) == T_FIXNUM) {
        args.append("\"");
        args.append(k);
        args.append("\"");
    } else {
        args.append(k);
    }
    args.append(""");
    proname.append(".call");
    proname.append(args.c_str());
    rb_eval_string(proname.c_str());
    return SOAP_OK;
"
```

Automatic generation of visualization web services

```ruby
int ns__rbmultipleWidgetsSetRepresentationMode (SOAP* soap, xsd__string mode, int cid,
    ns__rbmultipleWidgetsSetRepresentationModeResponse* result) {
    rbinterp->ensureRubyInitialized();
    string proname = "@setRepresentationMode";
    string args = "(";
    string t[2];
    VALUE val[2];
    rb_eval_string("str = /[a-zA-Z]="/);
    t[1].append("\"");
    t[1].append(mode);
    t[1].append("\" =\" str\";
    val[1] = rb_eval_string(t[1].c_str());
    if (TYPE(val[1]) == T_FIXNUM) {
        args.append("\"");
        args.append(mode);
        args.append("\"");
    } else {
        args.append(mode);
    }
    args.append("")
    proname.append(".call");
    proname.append(args.c_str());
    rb_eval_string(proname.c_str());
    return SOAP_OK;
}
```
Chapter B. Web methods generated for multipleWidgets.rb

```ruby
int ns_rbmultipleWidgetschangeObject (SOAP* soap,
    xsd_string i, int cid,
    ns_rbmultipleWidgetschangeObjectResponse* result) {
    rbinterp->ensureRubyInitialized();
    string proname = "@changeObject";
    string args = "(";
    string t[2];
    VALUE val[2];
    rb_eval_string("str = /[a-zA-Z]/");
    t[1].append("\n");
    t[1].append(i);
    t[1].append("str =\n");
    val[1] = rb_eval_string(t[1].c_str());
    if (TYPE(val[1]) == T_FIXNUM) {
        args.append("\n");
        args.append(i);
        args.append("\n");
    } else {
        args.append(i);
    }
    args.append(")\n");
    proname.append(".call\n");
    proname.append(args.c_str());
    rb_eval_string(proname.c_str());
    return SOAP_OK;
}

int ns_rbmultipleWidgetsrenWinback (SOAP* soap, int cid,
    ns_rbmultipleWidgetsrenWinbackResponse* result) {
    t5.start();
    char *stde;
    stde = new char;
    int sip = soap->ip; //Must assigned before sprintf
    sprintf(stde, "%d.%d.%d.%d", (sip >> 24)&0xFF, (sip >> 16)&0xFF, (sip >> 8)&0xFF, sip&0xFF);
    string cip;
    cip = stde;
    delete stde;
    bool findid = false;
    bool findip = false;
    for(int ipind = 0; ipind<cinf.size(); ipind++) {  //
        cinf is a global vector for store the
        information of client IP and socket
        //cout<< cinf.at(ipind).clientID <<"!!!"<<cinf.at(ipind).connectSocket<<endl;
        if (cinf.at(ipind).clientID == cid) {
            findid = true;
            if (cinf.at(ipind).clientIP == cip) {
```
findip = true;

t1.start();
rbininterp->ensureRubyInitialized();
VALUE ptr = rb_eval_string("@renWin");
t1.end();
    //if (!err){
    
    t2.start();
    rbVTKSpecialObject *obj;
    Data_Get_Struct(ptr, rbVTKSpecialObject, obj);
    void *ptr2 = obj->ptr;
    vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin = (vtkRenderWindow*)ptr2;

    renWin->Render();

    vtkWindowToImageFilter *w2i =
        vtkWindowToImageFilter::New();
    vtkPNGWriter *writer = vtkPNGWriter::New();
    w2i->SetInput( renWin );
    writer->SetInput( w2i->GetOutput() );
    writer->SetInputConnection (w2i->GetOutputPort());
    writer->WriteToMemoryOn();
    writer->SetFileName("RenderedImage.png");
    writer->Write();
    t2.end();
    t3.start();
    vtkUnsignedCharArray* image_object = writer->
        GetResult();
    int isize = image_object->GetNumberOfTuples();
    int i;
    unsigned char *img_data = image_object->
        Pointer(0);
    t3.end();
    int BytesSent = 0;
    int BytesIndex = 0;
    int BytesLeft = isize;
    t4.start();
    while( BytesIndex < isize ) {
        BytesSent = send(cinfo.at
            (ipind).connectSocket,
            &img_data[BytesIndex ], BytesLeft, 0);
        if(BytesSent<0) break;
        BytesLeft -= BytesSent;
BytesIndex += BytesSent;
    // cout << "Sent client "
    << BytesSent << " bytes" << endl;
}
t4.end();
    // cout << "Sent client " << isize << " bytes" << endl;
    char k[3];
    strcpy(k, "eof");
    for (int i = 0; i<3; i++)
        send(cinfo.at(ipind).connectSocket,&k[i],1,0);
    result->response = "IMAGE_OK";
t5.end();
ts5.dumpTimings();
    return SOAP_OK;
}/*}else {
    result->response = "Can not find the object for your render window, or it is incomplete!";
    return SOAP_OK;
}*/
}
}
if(findip == false){
    if(findid == false){
        result->response = "Invalid unique id, please verify your id!";
        return SOAP_OK;
    }
    result->response = "Can not find your image socket, try start an other client ";
    return SOAP_OK;
}
}
int ns_rbmultipleWidgetsrotationAngle(SOAP* soap, double a, xsd_string renwinName, int cid, ns_rbmultipleWidgetsrotationAngleResponse* result) {
    t5.start();
    char *stdc;
    stdc = new char;
    int sip = soap->ip; // Must assigned before sprintf
```c++
sprintf(stderr, "%%d.%%d.%%d.%%d", (sip >> 24)&0xFF,(sip >> 16)&0xFF,(sip >> 8)&0xFF,sip&0xFF);

string cip;
cip = stde;
delete stde;
bool findid = false;
bool findip = false;
for(int ipind = 0; ipind<cinfo.size(); ipind++){ // cinfo is a global vector for store the
    // information of client IP and socket
    //cout<< cinfo.at(ipind).clientID <<"!!!"<< cinfo.at(ipind).connectSocket<<endl;
    if(cinfo.at(ipind).clientID == cid){
        findid = true;
        if(cinfo.at(ipind).clientIP == cip){
            findip = true;
        }
    }
}

// call the rotate method with the angle and the render window object
 rotate(renWin, a);
return SOAP_OK;
```
int ns_rbmultipleWidgetsrotationGUI(SOAP* soap, double sizeImgX, double sizeImgY, double posFirstX, double posFirstY, double posLastX, double posLastY, xsd_string renwinName, int cid, ns_rbmultipleWidgetsrotationGUIService* result) {

t5.start();
  char *stde;
  stde = new char;
  int sip = soap->ip;  //Must assigned before sprintf
  sprintf(stde, "%%d.%%d.%%d\n", (sip >> 24)&0xFF,(sip >> 16)&0xFF,(sip >> 8)&0xFF,sip&0xFF);
  string cip;
  cip = stde;
  delete stde;
  bool findid = false;
  bool findip = false;
  for(int ipind = 0; ipind<cinfo.size(); ipind++){
    //cinfo is a global vector for store the information of client IP and socket
    //cout<< cinfo.at(ipind).clientID<<"!!!"<< cinfo.at(ipind).connectSocket<<endl;
    if(cinfo.at(ipind).clientID == cid){
      findid = true;
      if(cinfo.at(ipind).clientIP == cip){
        findip = true;
    }
  }

  //rotation angle calculation
  double startx = posFirstX - sizeImgX;
  double starty = posFirstY - sizeImgY;
  double starttan, startangle;

  if(startx < 0 & & starty >= 0){
    starttan = startx/starty;
    startangle = 90 + atan(fabs(starttan)) * 180 / M_PI;
  } else if(startx < 0 & & starty < 0){
    starttan = starty/startx;
    startangle = 180 + atan(fabs(starttan)) * 180 / M_PI;
  } else if(startx >= 0 & & starty < 0){
    starttan = startx/starty;
    startangle = 270 + atan(fabs(starttan)) * 180 / M_PI;
  } else{
    starttan = starty/startx;
    startangle = atan(fabs(starttan)) * 180 / M_PI;
  }

  double x = posLastX - sizeImgX;
  double y = posLastY - sizeImgY;
  double tang, stopangle;
}

Automatic generation of visualization web services
if (x < 0 && y >= 0) {
    tang = x / y;
    stopangle = 90 + atan(fabs(tang)) * 180 / M_PI;
} else if (x < 0 && y < 0) {
    tang = y / x;
    stopangle = 180 + atan(fabs(tang)) * 180 / M_PI;
} else if (x >= 0 && y < 0) {
    tang = x / y;
    stopangle = 270 + atan(fabs(tang)) * 180 / M_PI;
} else {
    tang = y / x;
    stopangle = atan(fabs(tang)) * 180 / M_PI;
}

// double angle = stopangle - startangle;
double angle = startangle - stopangle;

rbinterp->ensureRubyInitialized();
string rw = "@";
rw.append(renwinName);
VALUE ptr = rb_eval_string(rw.c_str());

rbVTKSpecialObject *obj;
Data_Get_Struct(ptr, rbVTKSpecialObject, obj);
void *ptr2 = obj->ptr;
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin = (vtkRenderWindow*)ptr2;

// call the rotate method with the angle and the render window object
rotate(renWin, angle);

return SOAP_OK;

}
```c
int ns__rbmultipleWidgetstranslation(SOAP* soap, double posLastX, double posLastY, xsd_string renwinName, int cid, ns__rbmultipleWidgetstranslationResponse* result)
{

t5. start();
    char *stde;
    stde = new char;
    int sip = soap->ip; //Must assigned before sprintf
    sprintf(stde, "%d.%d.%d.%d", (sip >> 24)&0xFF, (sip >> 16)&0xFF,(sip >> 8)&0xFF,sip&0xFF);
    string cip;
    cip = stde;
    delete stde;
    bool findid = false;
    bool findip = false;
    for(int ipind = 0; ipind<cinf.o.size(); ipind++) //
        cinfo is a global vector for store the //
            information of client IP and socket //cout<< cinfo.at(ipind).clientID<<"!!!"<<cinfo.at( //
            ipind).connectSocket<<endl;
    if(cinfo.at(ipind).clientID == cid){
        findid = true;
        if(cinfo.at(ipind).clientIP == cip){
            findip = true;
        
    t1. start();
    rbinterp->ensureRubyInitialized();
    string rw = "@";
    rw.append(renwinName);
    VALUE ptr = rb_eval_string(rw.c_str());
    t1. end();
    rbVTKSpecialObject *obj;
    Data__Get_Struct(ptr,rbVTKSpecialObject, obj);
    void *ptr2 = obj->ptr;
    vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin = (vtkRenderWindow*)ptr2;

    // call the rotate method with the angle and the render window object //
    translate(renWin, posLastX, posLastY);

    return SOAP_OK;
}
}
if(findip == false){
if(findid == false){
```

Automatic generation of visualization web services

54
result->response = "Invalid unique id, please verify your id!";
return SOAP_OK;
}
result->response = "Can not find your image socket, try start an other client ";
return SOAP_OK;
}

int ns_rbmultipleWidgetstranslationGUI(SOAP* soap,
    double sizeImgX, double sizeImgY, double posLastX,
    double posLastY, xsl_string renwinName, int cid,
    ns_rbmultipleWidgetstranslationGUIResponse* result) {

t5.start();
    char *stde;
    stde = new char;
    int sip = soap->ip; //Must assigned before sprintf
    sprintf(stde, "%d.%d.%d.%d", (sip >> 24)&0xFF,
            (sip >> 16)&0xFF,(sip >> 8)&0xFF,sip&0xFF);
    string cip;
    cip = stde;
    delete stde;
    bool findid = false;
    bool findip = false;
    for(int ipind = 0; ipind<cinfo.size(); ipind++)
    { // cinfo is a global vector for store the
       // information of client IP and socket
       // cout<< cinfo.at(ipind).clientID "!!!"<<cinfo.at(
       // ipind).connectSocket<<endl;
    if(cinfo.at(ipind).clientID == cid){
      findid = true;
    if(cinfo.at(ipind).clientIP == cip){
      findip = true;
    cout << "posLastX" : "<<posLastX<<endl;
    double newX = posLastX - sizeImgX;
    double newY = sizeImgY - posLastY;
    rbinterp->ensureRubyInitialized();
    string rw = "@";
    rw.append(renwinName);
    VALUE ptr = rb_eval_string(rw.c_str());
    rbVTKSpecialObject *obj;
    Data_Get_Struct(ptr, rbVTKSpecialObject, obj);
    void *ptr2 = obj->ptr;
    vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin = (vtkRenderWindow*)ptr2;
}
// call the translate method with the new position
translate(renWin, newX, newY);

return SOAP_OK;
}  

if(findip == false){
if(findid == false){
  result->response = "Invalid unique id, please verify your id!";
  return SOAP_OK;
}
result->response = "Can not find your image socket, try start an other client ";
return SOAP_OK;
}

int ns_rbmultipleWidgetszooming(SOAP* soap, double z,
  xsd_string renwinName, int cid,
  ns_rbmultipleWidgetszoomingResponse* result) {

t5.start();
  char *stde;
  stde = new char;
  int sip = soap->ip; //Must assigned before sprintf
  sprintf(stde, "%d.%d.%d.%d", (sip >> 24)&0xFF,(sip
  >> 16)&0xFF,(sip >> 8)&0xFF,sip&0xFF);
  string cip;
  cip = stde;
  delete stde;
  bool findid = false;
  bool findip = false;
  for(int ipind = 0; ipind<cinfo.size(); ipind++){
    // cinfo is a global vector for store the information of client IP and socket
    // cout<< cinfo.at(ipind).clientID <<"!!!"<< cinfo.at( ipind).connectSocket<<endl;
    if(cinfo.at(ipind).clientID == cid){
      findid = true;
    if(cinfo.at(ipind).clientIP == cip){
      findip = true;

t1.start();
  rbinterp->ensureRubyInitialized();
  string rw = ":@";
  rw.append(renwinName);
  VALUE ptr = rb_eval_string(rw.c_str());
Automatic generation of visualization web services
t1.end();
    rbVTKSpecialObject *obj;
    Data_Get_Struct(ptr, rbVTKSpecialObject, obj);
    void *ptr2 = obj->ptr;
    vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin = (vtkRenderWindow*)ptr2;

    zoom(renWin, z);

    return SOAP_OK;

    }

    }

    if (findip == false){
        if (findid == false){
            result->response = "Invalid unique id, please verify your id!";
            return SOAP_OK;
        }
        result->response = "Can not find your image socket, try start an other client ";
        return SOAP_OK;
    }

}
## Appendix C

### Script generated for the GUI

#### Listing C.1: Script Ruby generated for the GUI

```ruby
require 'tk'
require 'MyServiceDriver.rb'
require 'tkextlib/tkimg'
require 'socket'
include Socket::Constants

class MyService
  def id_back
    parameters = Idback.new
    idbackResponse = @obj.idback(parameters)
    @cid = idbackResponse.id_response
    @port = idbackResponse.port_response
    @sock = TCPSocket::new('127.0.1.1', @port)
  end

  def zoom
    if @renwin == "renWin" then
      parameters = RbmultipleWidgetszooming.new(@nbzoom, @renwin, @cid)
      resultZoom0 = @obj.rbmultipleWidgetszooming(parameters)
      parametersImg = RbmultipleWidgetsrenWinback.new(@cid)
    end
    cpt = 0
    eof = false
    @buf = []
    threads = []
    threads << Thread.new(0) do
      while true
        data = @sock.recvfrom(1)
        if data[0] == "e" then
          cpt = cpt + 1
        end
      end
    end
  end
end
```
elif data[0] == "o" and cpt == 1 then
cpt = cpt + 1
elif data[0] == "f" and cpt == 2 then
eof = true
else
cpt = 0
eof = false
end
@buf << data
break if eof == true
end
end
threads << Thread.new(1) do
  @resultImageback = @obj.rbmultipleWidgetsrenWinback(parametersImg)
end
threads.each{|thr| thr.join}
@time = Time.now.strftime("%I:%M:%S")
@logConsole.insert("@0, 0 ", "#{@time} -- Action performed : zoom => #{@resultImageback.response}\n")
if @resultImageback.response == "IMAGE_OK" then
  @str = Tk::BinaryString(@buf.slice(0, @buf.length - 4).to_s)
end
end
end

def rotation
if @renwin == "renWin" then
parameters = RbmultipleWidgetsrotationGUI.new(@sizex, @sizex, @startx, @starty, @stopx, @stopy, @renwin, @cid)
resultRotation = @obj.rbmultipleWidgetsrotationGUI(parameters)
parametersImg = RbmultipleWidgetsrenWinback.new(@cid)
cpt = 0
eof = false
@buf = []
threads = []
threads << []
threads << Thread.new(0) do
  while true
    data = @sock.recvfrom(1)
    if data[0] == "e" then
      cpt = cpt + 1
    elsif data[0] == "o" and cpt == 1 then
      cpt = cpt + 1
    elsif data[0] == "f" and cpt == 2 then
      eof = true
    else
      ...
73  cpt = 0
74  eof = false
75  end
76  @buf << data
77  break if eof == true
78  end
79  end
80  threads << Thread.new(1) do
81    @resultImageback = @obj.rbmultipleWidgetsrenWinback(
82      parametersImg)
83  end
84  threads.each{|thr| thr.join}
85  @time = Time.now.strftime("%I:%M:%S")
86  @logConsole.insert("
87    @time = @obj.rbmultipleWidgetsrenWinback(
88    (parameters)
89  end
90  end
91  end
92  def translation
93    if @renwin == "renWin" then
94      parameters = RbmultipleWidgetstranslationGUI.new(
95        @sizex, @sizey, @translatex, @translatey, @renwin, @cid)
96      resultTranslation0 = @obj.
97        rbmultipleWidgetstranslationGUI(parameters)
98      parametersImg = RbmultipleWidgetsrenWinback.new(@cid)
99      cpt = 0
100     eof = false
101     @buf = []
102     threads = []
103     threads << Thread.new(0) do
104       while true
105         data = @sock.recvfrom(1)
106         if data[0] == "e" then
107           cpt = cpt + 1
108         elsif data[0] == "o" and cpt == 1 then
109           cpt = cpt + 1
110         elsif data[0] == "f" and cpt == 2 then
111           eof = true
112         else
113           cpt = 0
114           eof = false
115         end
116         @buf << data
117         break if eof == true
118     end
119   end
120  end
121

Automatic generation of visualization web services
end
threads << Thread.new(1) do
  @resultImageback = @obj.rbmultipleWidgetsrenWinback
  (parametersImg)
end
threads.each{|thr| thr.join}
@endtime = Time.now.strftime("%I:%M:%S")
@logConsole.insert('@0,0', "#{@time} -- Action performed : translation => #{@resultImageback.response}\n")
if @resultImageback.response == "IMAGE_OK" then
  @str = Tk::BinaryString(@buf.slice(0, @buf.length - 4)).to_s
end
end
end

def rbmultipleWidgetsrenWinRendering
  @renwin = "renWin"
  parameters = RbmultipleWidgetsrenWinback.new(@cid)
  cpt = 0
  eof = false
  @buf = []
  threads = []
  threads << []
  threads << Thread.new(0) do
    while true
      data = @sock.recvfrom(1)
      if data[0] == "e" then
        cpt = cpt + 1
      elsif data[0] == "o" and cpt == 1 then
        cpt = cpt + 1
      elsif data[0] == "f" and cpt == 2 then
        eof = true
      else
        cpt = 0
        eof = false
      end
      @buf << data
      break if eof == true
    end
  end
  threads << Thread.new(1) do
    @resultRbmultipleWidgetsrenWin = @obj.rbmultipleWidgetsrenWinback(parameters)
  end
  threads.each{|thr| thr.join}
  @time = Time.now.strftime("%I:%M:%S")
  @logConsole.insert('@0,0', "#{@time} -- Action performed : Rendering object => #{

Automatic generation of visualization web services
61
Chapter C. Script generated for the GUI

```python
@resultRbmultipleWidgetsrenWin.response \n"
if @resultRbmultipleWidgetsrenWin.response == "IMAGE_OK"
    @str = Tk::BinaryString(@buf.slice(0, @buf.length - 4).to_s)
end
end

def rbmultipleWidgetssetColor
    parameters = RbmultipleWidgetssetColor.new(
        @nbrbmultipleWidgetssetColor0.value,
        @nbrbmultipleWidgetssetColor1.value,
        @nbrbmultipleWidgetssetColor2.value, @cid)
    resultRbmultipleWidgetssetColor = @obj.
        rbmultipleWidgetssetColorColor(parameters)
    @renwin = "renWin"
    parametersImg = RbmultipleWidgetsrenWinback.new(@cid)
    cpt = 0
    eof = false
    @buf = []
    threads = []
    threads << Thread.new(0) do
        while true
            data = @sock.recvfrom(1)
            if data[0] == "e" then
                cpt = cpt + 1
            elsif data[0] == "o" and cpt == 1 then
                cpt = cpt + 1
            elsif data[0] == "f" and cpt == 2 then
                eof = true
            else
                cpt = 0
                eof = false
            end
            @buf << data
            break if eof == true
        end
    end
    threads << Thread.new(1) do
        @resultRbmultipleWidgetsrenWin = @obj.
            rbmultipleWidgetsrenWinback(parametersImg)
    end
    threads.each{|thr| thr.join}
    @time = Time.now.strftime("%I:%M:%S")
    @logConsole.insert('[@0, 0 ', "#{time} -- Action performed : rbmultipleWidgetssetColor => '#{ @resultRbmultipleWidgetsrenWin.response }\n"
if @resultRbmultipleWidgetsrenWin.response == "IMAGE_OK" then
```

Automatic generation of visualization web services
```python
@str = Tk::BinaryString(@buf.slice(0, @buf.length - 4).to_s)
end

def rbmultipleWidgetschangeObject
parameters = RbmultipleWidgetschangeObject.new(
    @nbrbmultipleWidgetschangeObject0.value, @cid)
resultRbmultipleWidgetschangeObject = @obj.
rbmultipleWidgetschangeObject(parameters)
@renwin = "renWin"
parametersImg = RbmultipleWidgetsrenWinback.new(@cid)
cpt = 0
eof = false
@buf = []
threads = []
threads << Thread.new(0) do
while true
    data = @sock.recvfrom(1)
    if data[0] == "e" then
        cpt = cpt + 1
    elsif data[0] == "o" and cpt == 1 then
        cpt = cpt + 1
    elsif data[0] == "f" and cpt == 2 then
        eof = true
    else
        cpt = 0
        eof = false
    end
    @buf << data
    break if eof == true
end
end
threads << Thread.new(1) do
    @resultRbmultipleWidgetsrenWin = @obj.
rbmultipleWidgetsrenWinback(parametersImg)
end
threads.each{|thr| thr.join}
@time = Time.now.strftime("%I:%M:%S")
@logConsole.insert("@0, 0", "#{@time} -- Action performed : rbmultipleWidgetschangeObject => #{
    @resultRbmultipleWidgetsrenWin.response}
"
if @resultRbmultipleWidgetsrenWin.response == "IMAGE_OK" then
    @str = Tk::BinaryString(@buf.slice(0, @buf.length - 4).to_s)
end
end
```

Automatic generation of visualization web services
```ruby
parameters = RbmultipleWidgetssetRepresentationMode.
  new(@nbrbmultipleWidgetssetRepresentationMode0.
    value, @cid)
resultRbmultipleWidgetssetRepresentationMode = @obj.
  rbmultipleWidgetssetRepresentationMode (parameters)
@renwin = "renWin"
parametersImg = RbmultipleWidgetsrenWinback.new(@cid)
cpt = 0
eof = false
@buf = []
threads = []
threads << Thread.new(0) do
  while true
    data = @sock.recvfrom(1)
    if data[0] == "e" then
      cpt = cpt + 1
    elsif data[0] == "o" and cpt == 1 then
      cpt = cpt + 1
    elsif data[0] == "f" and cpt == 2 then
      eof = true
    else
      cpt = 0
      eof = false
    end
    @buf << data
    break if eof == true
  end
end
threads << Thread.new(1) do
  @resultRbmultipleWidgetsrenWin = @obj.
  rbmultipleWidgetsrenWinback (parametersImg)
end
threads.each{|thr| thr.join}
@time = Time.now.strftime("%I:%M:%S")
@logConsole.insert("@
{time} Action performed : rbmultipleWidgetssetRepresentationMode
=> #{@resultRbmultipleWidgetssetRepresentationMode.response}\n")
if @resultRbmultipleWidgetssetRepresentationMode.response == "IMAGE_OK"
  @str = Tk::BinaryString(@buf.slice(0, @buf.length - 4).to_s)
end

def initialize
  endpoint_url = ARGV.shift
  @obj = MyServicePortType.new(endpoint_url)
  @obj.options["protocol.http.receive_timeout"] = 360
  root = TkRoot.new
end
```

Automatic generation of visualization web services
t i t l e  " P l e a s e w a i t w h i l e t h e S e r v i c e i s c a l l e d "
}
root . m i n s i z e ( 5 0 0 , 5 0 )

menuBar = TkMenu . new
file = TkMenu . new ( menuBar )
ftypes = [ [ "PNG" , '*png' ] ]
file . add ( "command" , 'label' => "Save Image As ..." , 'command' => proc { 
filename = Tk . getSaveFile ( 'initialfile' => '*png' , 'filetypes' => ftypes )
if filename != ""
  f = File . new ( filename , "w")
f << @str
f . flush
f . close
end
})
file . add ( "command" , 'label' => "Quit" , 'command' => proc { 
  exit
})
menuBar . add ( 'cascade' , 'menu' => file , 'label' => "File" )
root . menu ( menuBar )

@renwin = TkVariable . new
display = TkFrame . new ( root )

@buf = []
image = TkPhotoImage . new
scene = TkCanvas . new ( display ) do
  height 600
  width 600
end
scene . pack ( 'fill' => 'both' , 'side' => 'top' )
t = TkImage . new ( scene , 300 , 300 , 'image' => image )

@nbzoom = TkVariable . new
@nbzoom = 1
scene . bind ( 'B4-ButtonRelease' , lambda { |e| 
  @nbzoom = (@nbzoom + 0.1);
  zoom
  image = TkPhotoImage . new ( 'data' => @str );
  t . configure ( 'image' => image )
})
scene . bind ( 'B5-ButtonRelease' , lambda { |e| 
  @nbzoom = (@nbzoom - 0.1);
  zoom
  image = TkPhotoImage . new ( 'data' => @str );
  t . configure ( 'image' => image )
}
@startx = TkVariable.new
@starty = TkVariable.new
@stopx = TkVariable.new
@stopy = TkVariable.new
@sizex = 300
@sizey = 300

scene.bind('3', lambda e |
    @startx = e.x
    @starty = e.y
})

scene.bind('ButtonRelease-3', lambda e |
    @stopx = e.x
    @stopy = e.y
    rotation
    image = TkPhotoImage.new('data' => @str);
    t.configure('image' => image)
})

@translatex = TkVariable.new
@translatey = TkVariable.new

scene.bind('ButtonRelease-1', lambda e |
    @translatex = e.x
    @translatey = e.y
    translation
    image = TkPhotoImage.new('data' => @str);
    t.configure('image' => image)
})

renderButtons = TkFrame.new(display)

rbmultipleWidgetsrenWinButton = TkButton.new(
    renderButtons){
    text 'rbmultipleWidgetsrenWin'
}.grid('row' => 0, 'column' => 0, 'padx' => 10, 'pady' => 10)

rbmultipleWidgetsrenWinButton.command{
rbmultipleWidgetsrenWinRendering;
;image = TkPhotoImage.new('data' => @str);
    t.configure('image' => image)
}

renderButtons.pack('fill' => 'both', 'side' => 'bottom')
display.pack('fill' => 'both', 'side' => 'right')

actions = TkFrame.new(root)
rbmultipleWidgetsssetColorFrame = TkFrame.new(actions)
rbmultipleWidgetssetColorElementsFrame = TkFrame.new(  
rbmultipleWidgetssetColorFrame )
@nbrbmultipleWidgetssetColor0 = TkVariable.new
TkEntry.new(rbmultipleWidgetssetColorElementsFrame,  
'textvariable' => @nbrbmultipleWidgetssetColor0)
.grid(  
'row' => 0,  
'column' => 0,  
'padx' => 10,  
'pady' => 10)

@nbrbmultipleWidgetssetColor1 = TkVariable.new
TkEntry.new(rbmultipleWidgetssetColorElementsFrame,  
'textvariable' => @nbrbmultipleWidgetssetColor1)
.grid(  
'row' => 0,  
'column' => 1,  
'padx' => 10,  
'pady' => 10)

@nbrbmultipleWidgetssetColor2 = TkVariable.new
TkEntry.new(rbmultipleWidgetssetColorElementsFrame,  
'textvariable' => @nbrbmultipleWidgetssetColor2)
.grid(  
'row' => 0,  
'column' => 2,  
'padx' => 10,  
'pady' => 10)

rbmultipleWidgetssetColorElementsFrame.grid(  
'row' => 0,  
'column' => 0,  
'padx' => 10,  
'pady' => 10)
rbmultipleWidgetssetColorButtonFrame = TkFrame.new(  
rbmultipleWidgetssetColorFrame )
rbmultipleWidgetssetColorButton = TkButton.new(  
rbmultipleWidgetssetColorButtonFrame)  
{  
'text' => 'rbmultipleWidgetssetColor',  
}.pack
rbmultipleWidgetssetColorButton.command{  
rbmultipleWidgetssetColor  
'image' = TkPhotoImage.new('data' => @str);  
t.config('image' => image) }

rbmultipleWidgetssetColorButtonFrame.grid(  
'row' => 0,  
'column' => 1,  
'padx' => 10,  
'pady' => 10)
rbmultipleWidgetssetColorFrame.pack(  
'fill' => 'both',  
'side' => 'top')
rbmultipleWidgetschangeObjectFrame = TkFrame.new(  
'actions'  
).pack(  
'side' => 'left',  
'expand' => 'true')
rbmultipleWidgetschangeObject0ListBoxBar = TkScrollbar.new(
    rbmultipleWidgetschangeObject0ListBoxFrame).pack(side='right', fill='y')
rbmultipleWidgetschangeObject0ListBox.yscrollbar(  
    rbmultipleWidgetschangeObject0ListBoxBar)
rbmultipleWidgetschangeObject0ListBox.insert("end", "Cylinder")
rbmultipleWidgetschangeObject0ListBox.insert("end", "Cone")
rbmultipleWidgetschangeObject0ListBox.insert("end", "Cube")
rbmultipleWidgetschangeObject0ListBox.insert("end", "Sphere")

rbmultipleWidgetschangeObject0ListBoxFrame.grid(row=0, column=0, padx=10, pady=10)
rbmultipleWidgetschangeObjectElementsFrame.grid(row=0, column=0, padx=10, pady=10)
rbmultipleWidgetschangeObjectButtonFrame = TkFrame.new(rbmultipleWidgetschangeObjectFrame)
rbmultipleWidgetschangeObjectButton = TkButton.new(
    rbmultipleWidgetschangeObjectButtonFrame){
        text = 'rbmultipleWidgetschangeObject '}
}.pack
rbmultipleWidgetschangeObjectButton.command{
    @nbrbmultipleWidgetschangeObject0.value = rbmultipleWidgetschangeObject0ListBox.curselction[0];
    rbmultipleWidgetschangeObject
        ;image = TkPhotoImage.new('data' => @str);
    t.configure('image' => image)
}

rbmultipleWidgetschangeObjectButtonFrame.grid(row=0, column=1, padx=10, pady=10)
rbmultipleWidgetschangeObjectFrame.pack(fill='both', side='top')
rbmultipleWidgetssetRepresentationModeFrame = TkFrame.new(actions)
rbmultipleWidgetssetRepresentationModeEltsFrame = TkFrame.new(
    rbmultipleWidgetssetRepresentationModeFrame)
@nbrbmultipleWidgetssetRepresentationMode0 = TkVariable.new
rbmultipleWidgetssetRepresentationModeRadioBFrame = TkFrame.new(
    rbmultipleWidgetssetRepresentationModeEltsFrame)
radioButtonWireframe = TkRadioButton.
new(rbmultipleWidgetssetRepresentationModeRadioBFrame)
{
text 'Wireframe'
variable @nbrbmultipleWidgetssetRepresentationMode0
value 'Wireframe'
}.grid(‘row’ => 0, ‘column’ => 0, ‘padx’ => 10, ‘pady’ => 10)

radioButtonWireframe .command{
@nbrbmultipleWidgetssetRepresentationMode0 .value = radioButtonWireframe .value;
}

radioButtonSurface = TkRadioButton.
new(rbmultipleWidgetssetRepresentationModeRadioBFrame)
{
text 'Surface'
variable @nbrbmultipleWidgetssetRepresentationMode0
value 'Surface'
}.grid(‘row’ => 0, ‘column’ => 1, ‘padx’ => 10, ‘pady’ => 10)

radioButtonSurface .command{
@nbrbmultipleWidgetssetRepresentationMode0 .value = radioButtonSurface .value;
}

rbmultipleWidgetssetRepresentationModeRadioBFrame .grid(‘row’ => 0, ‘column’ => 0, ‘padx’ => 10, ‘pady’ => 10)

rbmultipleWidgetssetRepresentationModeEltsFrame .grid(‘row’ => 0, ‘column’ => 0, ‘padx’ => 10, ‘pady’ => 10)

rbmultipleWidgetssetRepresentationModeButtonFrame = TkFrame.
new(rbmultipleWidgetssetRepresentationModeFrame)

rbmultipleWidgetssetRepresentationModeButton = TkButton.
new(rbmultipleWidgetssetRepresentationModeButtonFrame)
{
text 'rbmultipleWidgetssetRepresentationMode'
}.pack

rbmultipleWidgetssetRepresentationModeButton .command{
rbmultipleWidgetssetRepresentationMode
;image = TkPhotoImage.
ew(‘data’ => @str);
t .configure(‘image’ => image)
}

rbmultipleWidgetssetRepresentationModeButtonFrame .grid(‘row’ => 0, ‘column’ => 1, ‘padx’ => 10, ‘pady’ => 10)
rbmultipleWidgetssetRepresentationModeFrame.pack(
    'pady' => 10)
rbmultipleWidgetssetRepresentationModeFrame.pack(
    'fill' => 'both', 'side' => 'top')

logFrame = TkFrame.new(actions)
@logConsole = TkText.new(logFrame, 'height' => 5).pack
    ('side' => 'left', 'expand' => 'true')
bar = TkScrollbar.new(logFrame).pack('side' => 'right',
    'fill' => 'y')
@logConsole.yscrollbar(bar)
logFrame.pack('side' => 'bottom')
actions.pack('fill' => 'both', 'side' => 'left')

root.title = 'MyService'

Tk.mainloop

Automatic generation of visualization web services