SCRIPTING TOOLS FOR BEAMLINE COMISSIONING AND OPERATION

A. Pazos[#], S. Fiedler, EMBL-Hamburg, Hamburg, Germany
P. Duval, DESY, Hamburg, Germany

Abstract

Scripting tool capabilities are a valuable help for beamline commissioning and for advanced user operation. They are the perfect complement to static Graphical User Interfaces allowing one to create different applications in a rapid way. A light middle-layer for scripting support has been foreseen for the EMBL structural biology beamlines at the PETRA III synchrotron in Hamburg, Germany, to provide 'controlled' rather than 'direct' access to the control system devices. This prevents conflicts with the control system and allows control of the supported operations. In order to account for the wish of different scripting languages by the beamline scientists an extension of the scripting capabilities of the TINE control system has been implemented. To the existing shell support, a Python extension (PyTine) has been added and a Perl wrapping has been also prototyped (tine4perl). An explanation of these implementations and the different wrapping possibilities is also described in this paper.

INTRODUCTION

The EMBL-Hamburg outstation is commissioning three beamlines at the new PETRAIII light source at DESY (Hamburg). In addition, two beamlines at the DORIS storage ring are available for testing and prototyping the arriving instruments.

The control software is based on a client/server architecture integrated with the TINE control system [1]. Each device exports a TINE server that allows its remote operation. Flexibility has been a key feature since the design phase. For this reason different kinds of programming languages like C/C++, Python and Labview™ are supported.

The client side is mainly represented by Graphical User Interfaces (GUI) that connect themselves to the existing device servers. Two kinds of GUIs are available depending of the application. On one side there is an advanced control GUI that allows the operation and tuning of the entire beamline. This is mainly used by the beamline operators and experienced personnel. On the other side there is a GUI for visiting scientists with limited functionality with the main purpose of performing the data collection.

Some procedures, not even supported by the advanced GUI, need to be executed during the commissioning. Moreover, advanced users have the requirement of executing different strategies that are not foreseen at the user GUI.

In both situations the availability of a flexible and rapid way of executing this set of actions is very desirable. For this reason a scripting layer has been introduced at the software architecture allowing one to "glue" calls to the device servers. For gluing and system integration a scripting language can be 5-10 times faster than a system language [2] and the strong typing makes the programs easier to manage.

It is not desirable to the overall operation of a beamline that a user, not familiar with the installed hardware, is allowed to freely execute server functions. Of course, there are control system security measures, but overlaying the servers with a light scripting interface makes the system safer. Thanks to this scripting layer, the naming convention of the functions can be freely chosen.

SCRIPTING REQUIREMENTS

On the basis of our experience with beamline operation and after evaluating the specifications given by the beamline scientists, a list of requirements for the desired scripting environment was compiled:

- Easy to learn (for the developers and for the users)
- Easy to maintain
- Flexible (possible to refactor)
- Dynamic (does not need variable declarations)
- Well defined syntax
- Well documented
- Possible to control the accessible functionality
- Separated from the device specific layer
- Command-line support
- Sequencer support
- Reliable
- Secure
- User proof
- Multi-platform
- Open-source

TINE FOR SCRIPTING

The TINE control system originally supported a set-up of shell commands meant to build shell scripts both in Linux and Windows. Examples for these are the 'tget' (to receive data from a server) and the 'tput' (to send data to a server) commands. These functions are implemented in C and make use of the TINE C API. They receive as an input the necessary information (address, property, data type and data size) to make a call to a server.

[#]apazos@embl-hamburg.de

At first instance, they have been extensively used for commissioning and currently are used for setting up initialization scripts. For experienced users and developers, they allow efficient operation. . However, for users not familiar with the shell environment they might appear cumbersome.

Considering the defined requirement list and adding some extra valuable points (listed bellow), Python [3] was selected as the main supported scripting language.

- It has object oriented possibilities.
- Is getting more popular inside many scientific communities.
- It is also a powerful programming language.
- There are multiple open source libraries available.
- It is also possible to compile and to create executables.
- It is extendable and embeddable.
- There exists graphical support (PyQT [4] and others).
- There is already experience in our group.
- The GUI used at our MX beamline (MxCube [5]) is based on Python.

PYTINE

Initially there was no API for accessing the TINE control system from Python. First ideas where shown at the TINE Workshop, 2007 [6] demonstrating the possibility and the ease of performing such a task. With this starting point an evaluation of the different alternatives was performed.

Native implementation

A native implementation of the TINE control system in Python was evaluated. This would have meant a long term project with complex network implementations. It also would imply a big effort for maintaining and keeping it up to date. This possibility was beyond the scope of the project, having a TINE C API and taking into account that the most-widely used implementation of the Python programming language is written in C.

Python Bindings

The idea was to wrap the TINE C library, implementing Python bindings on top of this (see Fig. 1).

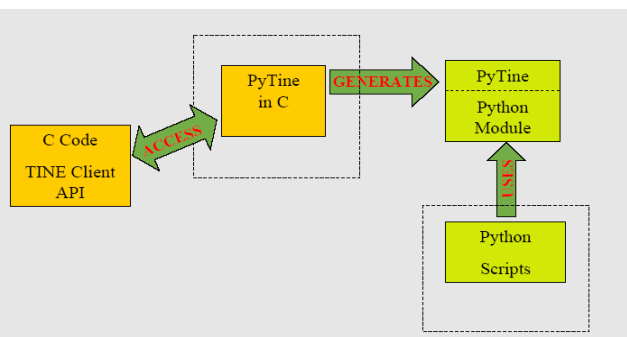

Figure 1 – PyTINE implementation overview

This concept had been successfully used for giving support to other programming languages, such as Labview™ and MatLab™. The use of the TINE Java library was discarded because of better experience of the developers with the C API. The desired outcome was a Python library totally transparent to the C interfaces.

The possibility of using a translator library was also tested. The most popular systems were installed and evaluated: Boost.Python [7] and Swig (Simplified Wrapper and Interface Generator) [8]. The Boost libraries turn out to support more functionality for Python and to be more extended than Swig, but in both cases the translation was not a fully automatic process. For this reason, a native binding inside the C code, without dependencies on a third part library, was decided. This was based on the direct use of the Python.h library and generated with a standard gcc compiler.

All the TINE client functionality was wrapped and a set of new functions was implemented in order to provide a generic friendly interface. This collection constitutes the PyTINE API and its main characteristics are:

- Callback capabilities.
- Support for the TINE data types.
- Data structures available.
- Tested in Linux and Windows.
- Plot functionality integrated, thanks to the use of the PyPlot library [9].
- Integrated inside Labview applications using LabPython [10].

Scripting Middle Layer

As mentioned in the previous section, the PyTINE library is not meant to be invoked directly by the user scripts. It is imported by a set of Python modules, provided by the developers, which create the available functions for implementing scripts. Each of these modules have a specific functionality attached to one or more device servers. They are implemented following an object oriented approach.

In order to perform for example a non-standard data collection, a user can easily implement a Python script. This will call the supported methods of the dataCollection class, which internally take care of the correct operation (see Fig. 2).

```
import dataCollection

//set the exposure parameters
prefix = tst1
dir = /home/marccd/images
run = 1
distance = 320
startphi = 0
phirange = 1.0
exposure = 1.0
frames = 10

# move distance to start synchronous
dataCollection.moveDistance(325)

# start data collection
i = 1
```

```
while i <= frames:
  status = dataCollection.exposeFrame(PHI,
exposure, startphi, phirange, run, dir ,prefix)
 print "Exposing frame ", i , " result: ",
status
 i++

print "Data Collection Finished"
```

Figure 2 – Script to set parameters like rotation angle, starting angle for the phi axis of a diffractometer, detector to crystal distance and initialize a rotating crystal data collection with Xrays recorded by a CCD area detector

TINE4PERL

After the implementation of PyTINE the possibility of interfacing TINE with Perl [11] was also tested. The target was to get and put synchronous data of the basic data types. Making use of the flexibility and extensibility of the control system it is possible to accommodate different developer's flavours regarding programming languages.

Thanks to the experience acquired with the prior implementations, this turned out to be a minor task. Because only the basic functionality was needed and the good support provided for Perl [12], the Swig translation library was selected. To do this, a SWIG interface file (with the extension .i) had to be written. In this file, the ANSI C prototypes that have to be accessed from Perl are listed. In addition, some SWIG directives had to be included. In our case, some specific functions to treat arrays and strings were implemented. Invoking the SWIG command two files are produced: the tine4perl_wrap.c, which contains the C wrapper functions and the tine4perl.pm, which contains the supporting Perl code needed to load and use the module. As last step, the wrapped file has to be compiled and linked into a shared library (see Fig. 3).

```
INCL = /usr/include/tine/
LIBS = /usr/lib

CPP = g++ -fPIC -shared
CC = gcc -g -fPIC -Wall -I${INCL} -c
CCL = cc -g
LM = -lm
LD = ld -G
SWIGPERL = swig -perl5
CCPERL = gcc -I${INCL} -c

tine4perl.so: tine4perl.o
        ${LD} tine4perl.o tine4perl_wrap.o
${LIBS}/libtinemt.so -o tine4perl.so

tine4perl.o: tine4perl.c
        ${SWIGPERL} tine4perl.i
        ${CCPERL} tine4perl.c tine4perl_wrap.c
`perl -    MExtUtils::Embed -e ccopts`
```

Figure 3 – TINE4PERL 'make' commands. It uses a standard gcc compiler and the generated objects to the multithread tine library

CONCLUSION AND OUTLOOK

A scripting language is suited to perform different tasks than a system programming language. We have seen in our applications that if they are used together they can create very powerful programming environments fulfilling complementary requirements.

A scripting language should be as simple as possible. In some occasions it is beneficial not to provide a direct access to the system but to use a middle layer controlling the access to the device servers.

It is important to evaluate very carefully the existing wrapping solutions, including automatic converters, in order to support a new scripting language. Depending on the desired functionality it might be better to use one method or the other. On the one hand, the use of an automatic converter for complex implementations, that possibly include pointers and data structures, it can prove to be a tedious task, making it necessary to learn a special syntax. On the other hand, an automatic converter can create fast bindings for simpler wrappings.

In our environment, where all the software is integrated in a control system, flexible and open systems allow us to extend their functionality and to support new programming languages.

This scripting concept and architecture developed to control synchrotron beamlines could be extended and applied to different instrumental environments and integrated with different control systems.

REFERENCES

[1] P. Bartkiewicz and P. Duval, "TINE as an accelerator control system at DESY", Meas Sci Technol, 18:2379–2386, 2007, p. 2379-2386

[2] J. Ousterhout, "Scriptiong: Higher Level Pogramming for the 21st Century", IEEE Computer magazine, March 1998

[3] Python Programming Language, www.python.org

[4] PyQt White Paper, www.riverbankcomputing.com

[5] J. Gabadinho et al., "MxCuBE: a synchrotron beamline control environment customized for macromolecular crystallography experiments", J. Synchrotron. Rad., 2010, 17, 700-707

[6] D. Franke, "TINE+Python Bindings", (see http://tine.desy.de TINE Workshop 2007).

[7] C++ Boost Libraries, http://www.boost.org/

[8] Simplified Wrapped and Interface Generator (SWIG), http://www.swig.org/

[9] Matplotlib, http://matplotlib.sourceforge.net/

[10] Labpython, Open Source Python tools for Labview™, http://labpython.sourceforge.net/

[11] Perl Programming Language, http://www.perl.org/

[12] D. Beazley et al., "Perl Extension Building with SWIG", O'Reilly Perl Conference 2.0, 1998, San Jose, California