SCRIPTING LANGUAGES FOR SCIENTIFIC SOLUTIONS

Telpinskiy K., Ginkel M., Svyatniy V.A.
Max-Plank-Institute (Magdeburg), DonNTU

Abstract

Teplinskiy K., Ginkel M., Svyatniy V.A.. Scripting languages for scientific solutions. *Motivation: Realistic scientific tasks require a certain set of features from modern simulation environments. One of the most progressive methods for construction of the application structure is using an object-oriented approach for the application and allow for the control of the application with a scripting language. Results: In this article we examine advantages of this method, new approaches for interaction of scripting languages with an object-oriented CORE of the application and approaches of using scripting languages together with the graphic user interface. Examples of using Python language and main integration technologies with C ++ are resulted.*

Most modern applications are developed in object-oriented style and include a scripting language. The question is why? Seemingly, all algorithms can be realized beforehand, without using additional scripting languages, but in practice, this doesn't seem to work . For scientific solutions like modeling and simulation of process models, there exists a variety of different algorithms and realizations. Models and general solving algorithms can be standardized and setup by changing some parameters. But in many cases it's necessary to use combinations of different basic algorithms (e.g. equation solvers, integrators) to find useful solutions, especially if you are working with processes in different scientific fields. Every user has therefore own peculiarities, models, and develops solutions with different solving algorithms. Furthermore the number of useful basic algorithms is growing constantly. If computer programmers had to implement all the different combinations of algorithms in the general simulation software, it would be devastating and unmaintainable. On the other hand if the users would have to control the software by single actions on a GUI their work would be very inefficient. This problem is absolutely solved by the integration of a scripting language. In this case the programmer don't need to develop all possible combinations of software parts because users can implement this themselves with the scripting language at application runtime. It becomes unnecessary to constantly recompile and reinstall the project. User-defined algorithms are located in script files, and every user can work with a limited set of scripts for the work with his models. The only disadvantage is some loss of

performance but usually the more expensive parts in simulation are the basic numerical algorithms. This question we'll examine later.

What are scripting languages? According to their destination they have simple constructs and a comprehensible set of commands. The understanding of the source code should be easy and the user should be quickly able to perform simple tasks in it. Another important property of scripting languages is that they are mostly orientated to their application domain (for example the build-in language of Matlab). Scripting languages also belong to class of interpreters, it means that there is a possibility of executing scripts step by step, and to execute separate commands in command line mode, giving the user a handy interface and making debugging easier.

In the design of a new version of the DIVA simulation environment (nDIVA), Our task was to realize a scripting language, which should:

- have access to all numerical methods and data of nDIVA
- realize batch execution files and a command line interface for users
- make it possible to call scripts from the numerical methods (e.g. for a partial or full description of models in the scripting language).
- design a set of commands well suited for the application domain.

The first question was to choose tools and realization method for the scripting language. I started my investigations with analyzing the possibility of making my own scripting language. There are different methods for making it. But all of them boil down to a description of all lexical units and allowed constructs of the designated language grammar bottom up and to the realization of their semantics.

The starting point is to describe all regular expressions. Regular expressions are sets of symbols, numeric, union or alternation of them, which create lexical units.

For example:

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<small_char> ::= a | b | .. | y | z

But regular expression cannot describe nested constructs, that's why we should use a context-free grammar and make description in Backus-Naur Form (BNF).

BNF can be used not only for expression description, but also for the description of simple and compound operators. It gives the possibility to describe practically all existent-programming languages.

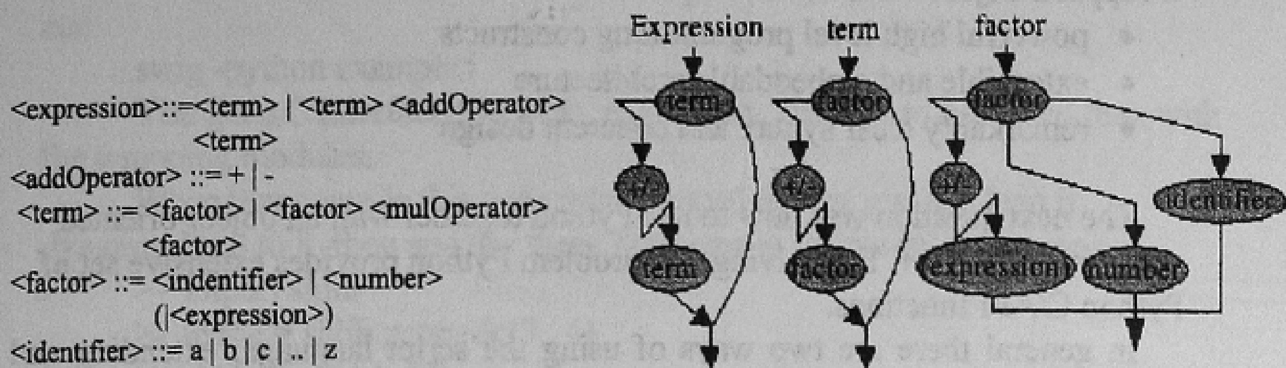

Figure 1. BNF description example (fragment of assignment operator)

The next problem, which I had to solve, was the following – what kind of parsing algorithms should be used. There are two main possibilities: Top-down parsers (LL) and Bottom-up (LR). LR parsing is based on shifting tokens on a stack during recognition. LL parsing is based on recursive subroutines. I've chosen the second one and realized it. The method is really simple: all terms are organized as separate functions and call each other recursively and all of them return error status (0) or number of characters used by this term (> 0). If there are no more nested terms, the functions will be exited one by one. If syntax is not correct the top function will get not the correct size of parsed source.

It really works properly but the problem is that it'll take you a lot of time to describe and program all language constructs. It is necessary to program identifiers, errors and decide all problems with memory allocation. It's really not so easy to make your own language because you should provide mechanisms for e.g. exceptions and type casting. The main problem is that it's very hard to test all elements and operation we used inside.

One possible variant I looked through and made examples is using standard features of the OS Linux, using lexical and syntax analyzers LEX and YACC. This feature gives possibility to make parsing of documents and make simple programming languages. For using them you have to write files with lexical and syntax (essentially machine-readable BNF) description elements and you define connection between description and operations. Those are rather good features, but they are not applicable for making complicated interpreters because their generated code is hardly modified and partial parsing is not supported.

After discussions we decided to use a standard interpreter which has good capabilities for integration with C++ CORE and has been tested to maturity. Now, in the modern software market there are a lot of good interpreters oriented for solving some task or for common use. The most popular are JScript, Perl, Python, VBScript and others. Perl is rather huge and complicated. VB is mostly used for automatization of Windows applications. We decided to use Python because:

it supports object-oriented development

- powerful high level programming constructs
- extensible and embeddable architecture
- remarkably clear syntax and coherent design

The next question was how to use Python together with an object-oriented core written in C++. For solving this problem Python provides extensive set of Python C/API functions.

In general there are two ways of using the script language: extending and embedding. Another name for extending is dynamic loading. If you decide to use it this way you should compile dynamically loaded libraries (shared library of Linux). You have to define a structure, which contains list of names and pointers to functions and a function for module initialization.
This is a small example:

```
#include "Python.h"
static PyObject * fn1(PyObject *self, PyObject *args)
{
      int mode;
      int oflags;
      int nofork;
      if (!PyArg_Parse(args, "(ii)", &oflag, &nofork))
            return NULL;
      ...
      return Py_BuildValue("(i)", "mode", mode);
}
static PyMethodDef sgi_methods[] = {
      {"getmode",       fn1},
      {NULL,            NULL}             /* sentinel */
};
void initshlib(void)
{
      Py_InitModule("shlib", sgi_methods); }
```

This wrapping can be done automatically by using special tools. We used SWIG (developed at Chicago University). It's a special software development tool for interfacing different languages with C and C++. Actually you can only call C functions from Python. But SWIG gives the possibility to use variables, constants, classes, exceptions, templates, exceptions, make type casting, and to extend wrapping by yourself (For a example we made a callback function mechanism using the Python API). All this is possible because SWIG wraps everything to functions and uses a mechanism of shadowing classes (Using C++ classes like original Python classes in scripts). You can also make emulations of container types. For using SWIG you just have to make SWIG interface file

where all elements and special features are described by a special syntax and run:

```
swig -python example.i
```

You get a C interface file wrap_example.c. This has to be linked then with the wrapping modules.

When you compile this code into a shared library you can load it dynamically into Python and use these functions in the modules namespace.

```
>> import shlib
>> mode = shlib.getmode(3 , 4)
>> shlib.showticks(7)
```

For using the shadowing of classes you should not import the shared library directly but use the SWIG generated file example.py instead. This module loads the shared library and provides the shadowing.

The other possibility is to use embedding of Python. It's very similar to static linking. By this method you can rebuild the scripting language as an interpreter with extensions. You use your programs main function and include all modules you want to use. This program adds all functions to a list (the same way as dynamic loading) and starts the interpreter. In this case we have to link the program with all Python libraries and get a new interpreter.

The question is in which way we use it. On one hand if you use dynamic loading your program it's just a set of shared libraries, Python libraries not included. It's smaller and any way these libraries can be used by another interface program or some ORB interface (if somebody prefer this way) at the same time. Statically linked applications are rather big because of the included Python libraries but can combine Python shell with any interfaces inside the program. And actually you don't need Python installed to user's PC, but provide it within your software.

Another question is the way of using Python together with a GUI. In our situation we had idea to make a server variant which provides description of models through Internet HTTP protocol, making simulation in the server and returning results of simulation. When you use some script language like Python, Perl and etc., it really convenient variant for Web solution, because most of Web Servers support executing CGI scripts or script embedding. Python is mostly recommended for use with the Zope Web Server (because Zope supports Python). As you can see on figure 1.2 in this case you can have remote access though HTTP protocol of Zope Web Server to some Python scripts which provide loading model, making simulation and getting result in HTML pages. Of course we should have HTML(DHML, PHP, etc) user interface and the server solves all problems with access administration.

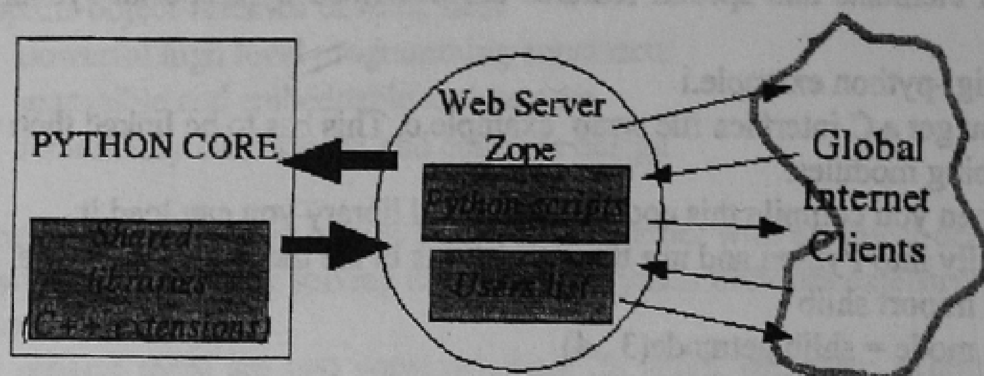

Figure 2. Web server variant is based on Python language

There are a lot of possibilities of using scripting language. For some tasks it's enough to extend Python by C++ modules and that's all, but for big projects it's not so, the scripting language will be used in parallel or integrated with an GUI. I see 2 basic variants. First of all we can use Python and the GUI separately. This variant is most convenient for the situation when only advanced users use the language, and beginners use only the GUI. For realization of this we have to compile modules as dynamically loaded libraries and load then from Python or from the GUI (Figure 3). If we use a CORBA interface we should build a "communicator" which loads libraries and provides them through CORBA to the GUI. Another way is to compile Python libraries and CORE statically(Figure 4). In CORE we have to provide access to modules which can be dynamically loaded or statically linked. It means that some useful module can be linked to CORE and be used in Python, another can be loaded dynamically. In our task models and solver are compiled to shared libraries and have to be loaded on the fly but there are some classes which are provided for common use. They are compiled them into the CORE and attached to Python when the CORE is executed. Of course we have to supply some "interface communicator" which provides information exchange between Python and the GUI.

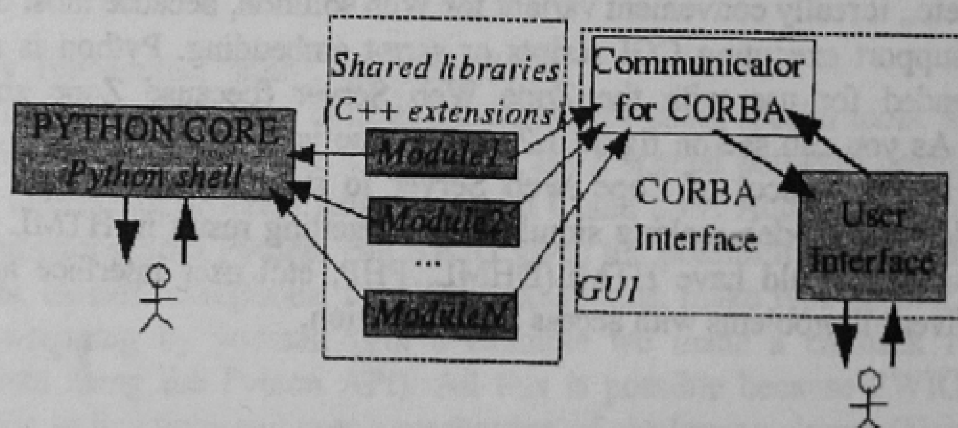

Figure 3. Application structure in the case of separated using GUI and Python shell

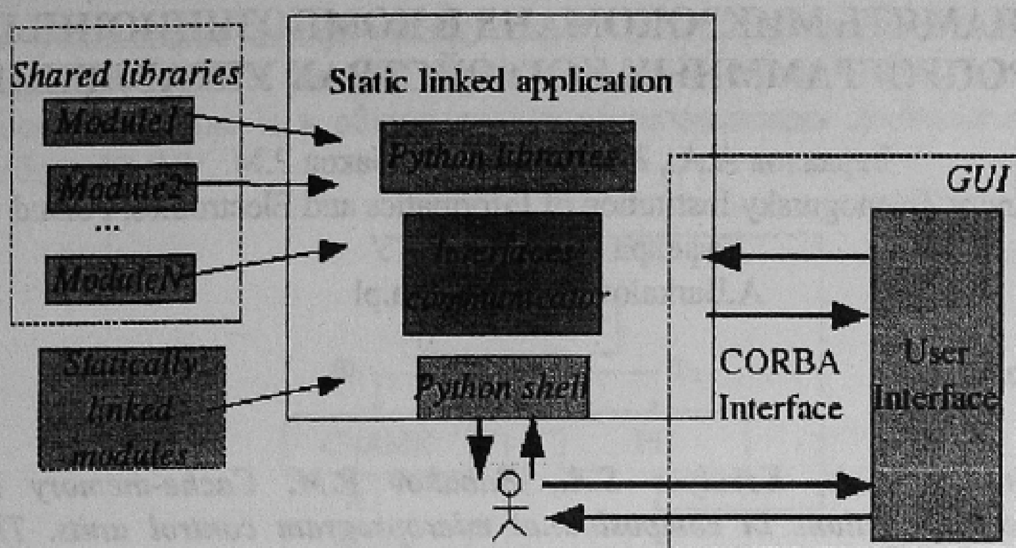

Figure 4. Application structure in the case of statically linking of Python and communication with external GUI

Finally, We want to note, that we have wide set of tools which give possibility to incorporate and fully use script languages like powerful features not only for scientific application, but business and any kinds of software(GIMP for example) application. Information in this article just shows different possibilities and advantages, but for different special cases this question is open.

REFERENCES

[1] M. Häfele, A. Kienle, E. Klein, A. Kremling, C. Majer, M. Mangold, A. Spieker, E. Stein, R. Waschler, K.P. Zeyer, User Manual DIVA - 3.9 Institut für Systemdynamik und Regelungstechnik Universität Stuttgart

[2] Mark Lutz, Programming Python, O'Reilly, October 1996, 1

[3] Devid W. Barron (University of Southampton), The world of scripting languages, John Wiley & Sons, Ltd, 2000

[4] D.M. Beazley, Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations, presented at Supercomputing'96, 1996

[5] Official Python(www.python.org) and SWIG(www.swig.org) web sites, including Python and SWIG documentation

[6] Mark Mitchell, Jeffrey Oldham, and Alex Samuel, Advanced Linux Programming, New Riders Publishing, First Edition, June 2001

[7] B.J. Keller, Programming languages, Department of Computer Science, Virginia Tech, 2000

[8] Mailing lists of Python (python-list@python.org) and SWIG(swig@cs.uchicago.edu) projects.

Дата надходження до редколегії: 25.12.2003 р.