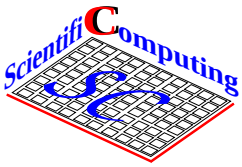

Calling a C++ CTL Component from Python Using SWIG under Linux/Unix

Martin Krosche

Version 1.0 (2005-07-22)

Location
Institute of Scientific Computing
Technical University Braunschweig
Hans-Sommer-Strasse 65
D-38106 Braunschweig

Postal Address
Institut für Wissenschaftliches Rechnen
Technische Universität Braunschweig
D-38092 Braunschweig
Germany

Contact

Phone:	+49-(0)531-391-3000
Fax:	+49-(0)531-391-3003
E–Mail:	`wire@tu-bs.de`
www:	`http://www.tu-bs.de/institute/WiR`

Introduction

The Component Template Library (CTL) is a middleware based on C++ generic template programming to realise distributed component-based software systems. This component technology is implemented with respect to best performance and also comfortable usage similar to CORBA or Java RMI.

Programming in Python [1] increases productivity and reliability [3]. Hence, not surprisingly, Python has entered the region of computational science. But Python is an interpreted language and therefore less high-performance than compiled languages like C++.

Thus it make sense to couple Python and the CTL. This manual shows how to call a C++ CTL component from Python. For this purpose, the wrapper code generator SWIG is used because of its comfort, richness in features and good documentation [2, 3].

Cooking Recept

The cooking recept to call a C++ CTL-component from Python under Linux/Unix consists of six steps.

1. create a CTL component from C++ implementation
2. create a front-end header for the CI
3. create a SWIG wrapper code interface
4. generate wrapper code
5. compile and link to shared library
6. call CTL component from Python

In the following the recept is mapped onto a simple C++ example class. The simple generation of the CTL component from C++ implementation is not shown in this manual, therefor see [].

The C++ example class `Add` contains methods to add scalars and vectors, see `add.h` and `add.cc`. The corresponding CI is given in `add.ci`. Some lines in `add.ci` are presented as continued lines by using the macro "\". The compiler's preprocessor removes such a backslash and combines correponding lines into one long line. Some compilers do not support this macro. Thus it is more portable to write one long line.

```
/* add.h */

#ifndef __ADD_H
#define __ADD_H
```

```
#include <vector>

namespace wire {
    class Add {
    public:
        Add();
        ~Add();

        int add( const int& x, const int& y, int& res );
        int add( int x, int y );
        void add( const std::vector<int>& x,
                const std::vector<int>& y,
                std::vector<int>& res );
        std::vector<int> add_v(
                const std::vector<int>& x,
                const std::vector<int>& y );
    };
}

#endif //_ADD_H


/* add.cc */

#include "add.h"

namespace wire {
    Add::Add() {}
    Add::~Add() {}

    void Add::add( const int& x, const int& y,
            int& res ) {
        res = x + y;
    }

    int Add::add( int x, int y ) {
        int res;
        add( x, y, res );
        return res;
    }

    void Add::add( const std::vector<int>& x,
```

```
            const std::vector<int>& y,
            std::vector<int>& res ) {
        res.resize(x.size());
        unsigned int i;
        for ( i = 0; i < x.size(); i++ ) {
            res[i] = x[i] + y[i];
        }
    }

    std::vector<int> Add::add_v(
            const std::vector<int>& x,
            const std::vector<int>& y ) {
        std::vector<int> res;
        add(x, y, res);
        return res;
    }
}


/* add.ci */

#ifndef __ADD_CI
#define __ADD_CI

#include <ctl.h>

#define CTL_Library Arith
#include CTL_LibBegin
    #define CTL_Class AddCI
    #include CTL_ClassBegin

        #define CTL_Method1 void, add, ( const int4, \
                const int4, int4 ), 3
        #define CTL_Method2 int4, add, ( const int4, \
                const int4 ), 2
        #define CTL_Method3 void, add, ( \
                const array<int4>, const array<int4>, \
                array<int4> ), 3
        #define CTL_Method4 array<int4>, add_v, ( \
                const array<int4>, const array<int4>), 2

    #include CTL_ClassEnd
```

```
#include CTL_LibEnd

#endif //__ADD_CI
```

In class `Add` three methods named `add` exist. Generally function overloading is supported by the CTL. But in our case the third method contains arguments of the abstract type `array`. Such an `array` can be mapped onto a `vector`, `set` or similar containers. Assuming two methods exist which arguments only differ in their types and the abstraction of these types is `array`, then the compiler is not able to resolve `array`. As a consequence such an abstract type has to be concretised in case of method overloading. This is done in `connect.h` by defining a concretised structure and passing it to method `connect`.

```
/* connect.h */

#define CTL_Connect

#include <add.ci>
#include <vector>
#include "add.h"

struct detail {
    CTL_Method(3, void, wire::Add::add,(
            const std::vector<int>&,
            const std::vector<int>&,
            std::vector<int>& ), 3);
};

void CTL_connect() {
    ctl::connect<Arith::AddCI, wire::Add, detail>();
}
```

Till yet nothing is done linked to the possibility of calling a CTL component from Python. Hence, the essential will be considered. Because SWIG is not able to handle CI constructs, a C++ front-end header has to be defined for the CI, see `add_fe.h`.

```
/* add_fe.h */

#ifndef __ADD_FE_H
#define __ADD_FE_H

#include <vector>
```

```
namespace Arith {
    class AddCI {
    public:
        void add( int& x, int& y, int& res );
        int add( int x, int y );
        void add( const std::vector<int>& x,
                const std::vector<int>& y,
                std::vector<int>& res );
        std::vector<int> add_v(
                const std::vector<int>& x,
                const std::vector<int>& y );
    };
}

#endif //_ADD_FE_H
```

SWIG requires a specific interface to make generating of wrapper code possible. This wrapper interface is defined in `add_comp.i`.

```
/* add_comp.i */

%module add_comp
%{
#include "add.ci"
%}

%include "typemaps.i"
%apply int *INPUT  int& x, int& y ;
%apply int *OUTPUT  int& res ;
%include "std_string.i"
%include "std_vector.i"
namespace std
{
    %template(IntVector) vector<int>;
}
%include "add_fe.h"

%extend Arith::AddCI
{
    static void use(std::string loc)
    {
        ctl::link p(loc);
```

```
            Arith::AddCI::use(p);
        }
    }
```

All instructions with leading % are SWIG specific. The construct `%module` contains all C++ header files necessary to compile the interface. Because references and pointers are not supported in Python, the interface handles such differences. In our case only references has to be wrapped but wrapping of pointers leads to the same act. First SWIG interface `typemaps.i` has to be included. Second all reference declarations in `add_fe.h` has to be replaced by using SWIG instruction `%apply`. For instance `int& x` is an input parameter and therefore mapped onto `int *INPUT`. Next `std_string.i` is included to support usage of C++ standard string. A string is not instanciated in `add_fe.h`, however `std::string` is needed later. Instances of type `std::vector` are wrapped by using the construct `namespace std`. After defining all these SWIG instructions the front-end header can be included. A CTL component can be realised as a thread, shared library or remote executable. Somehow or other to call a CTL component its location has to be known. All these properties have to be coded into one C++ standard string which is used to call a CTL component. Thus extensions of front-end class `AddCI` are required. To avoid modifying `add_fe.h`, extensions are encapsulated in wapper's interface by using construct `%extend` and defining method `use`.

Now SWIG is able to generate wrapper code and to link all together to a shared library which is visible for Python. The Shell-script `make_modules.sh` contains all required commands.

```
# make_modules.sh #

swig -python -c++ -I../ci add_comp.i
g++ -g -DCTL_Lib -I../ctl/include -I../ci \
    -I/usr/include/python2.3 -c add_comp_wrap.cxx
g++ --shared add_comp_wrap.o ../ctl/lib/libctlD.a \
    -o _add_comp.so
```

The first command instructs SWIG to create wrapper code. SWIG generates C++ wrapper code `add_comp_wrap.cxx` and Python module `add_comp.py`. `add_comp.py` is Pythons front-end module to call desired library. While generation warning

```
../ci/add_fe.h:24: Warning(509): Overloaded add(int,int)
is shadowed by add(int &,int &,int &) at
../ci/add_fe.h:23.
```

occures. That means, SWIG is not able to wrap the second C++ method `add` because wrapped method would equal already wrapped method of first C++ method `add`. The second command instructs the GNU C++ compiler to create

Bytecode `add_comp_wrap.o`. The third command generates the desired library `_add_comp.so` from wrapper's bytecode and CTL library `libctlD.a`. The leading "_" of `_add_comp.so` is needed because this name is used already in `add_comp.py` to call desired library.

Yet, all is done. Only `add_comp.py`, `_add_comp.so` and the CTL component, given by `service.exe`, are required to call the component from Python. The Python module `run.py` realises a small test environment.

```
# python run.py #

from add_comp import *
a = AddCI()
AddCI.use('../build/linux/service/service.exe')

print '# s = add(s_1, s_2)'
s_1 = 2; s_2 = 3
s = a.add(s_1, s_2)
print 's = %f' % s;

size = 3
v_1 = IntVector(size)
v_2 = IntVector(size)
for i in range(0,size):
    v_1[i] = 1; v_2[i] = 10
v = IntVector(size)
print '# a.add(v_1, v_2, v)'
a.add(v_1, v_2, v)
for i in v:
    print i

print '# v = add_v(v_1, v_2)'
v = a.add_v(v_1, v_2)
for i in v:
    print i
```

Executing

```
python run.py
```

in the command line interpreter delivers

```
# s = add(s_1, s_2)
 *** running service.exe as process 1 on munin ***
```

```
 *** working dir is ../build/linux/service ***
 *** last compiled Jul 15 2005 09:16:55 ***
s = 5.000000
# a.add(v_1, v_2, v)
11
11
11
# v = add_v(v_1, v_2)
11
11
11
 *** ../build/linux/service/service.exe
(tcp:134.169.77.185:13684:1:2) terminated ***
```

Method `AddCI.use` gets path to C++ CTL component `service.exe`. Our CTL service is run at the beginning and terminated cleanly at the end. As you can see, calling of first C++ method from Python differs with respect to C++ calling. Here, an output parameter is returned. This is the price of wrapping. In contrast with it, calling of the third method `add` does not differ. Thus, output references are handled similar in Python as in C++ in case of vectors.

Literatur

[1] Python Software Foundation. Python website and mail system. http://www.python.org/.

[2] SourceForge. Swig. http://www.swig.org/.

[3] T.J.Barth, M.Griebel, D.E.Keyes, R.M.Nieminen, D.Roose, and T.Schlick. *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, 2004.