PIK Report

No. 67

COUPLING DISTRIBUTED FORTRAN APPLICATIONS USING C++ WRAPPERS AND THE CORBA SEQUENCE TYPE

Thomas Slawig

POTSDAM INSTITUTE
FOR
CLIMATE IMPACT RESEARCH (PIK)

Author:
Dr. Thomas Slawig
TU Berlin (MA 6-2), Strasse des 17. Juni 136, 10623 Berlin
E-mail: slawig@math.tu-berlin.de

Herausgeber:
Dr. F.-W. Gerstengarbe

Technische Ausführung:
U. Werner

POTSDAM-INSTITUT
FÜR KLIMAFOLGENFORSCHUNG
Telegrafenberg
Postfach 60 12 03, 14412 Potsdam
GERMANY

Tel.: +49 (331) 288-2500
Fax: +49 (331) 288-2600
E-mail-Adresse: pik-staff@pik-potsdam.de

POTSDAM, DEZEMBER 2000

Abstract

The CORBA standard lacks a mapping of its Interface Definition Language IDL to FORTRAN which still is an important language of many scientific and engineering applications. Using the given IDL to C++ mapping and a wrapping technique an effective coupling of FORTRAN codes can be realized that in most cases avoids data copying. A way to use the IDL sequence type for the data transfer is shown which allows highly flexible interfaces.

1 Introduction

The Common Object Request Broker Architecture (CORBA) standard [1] with its Interface Definition Language (IDL) provides an easy and efficient way to define interfaces for distributed components written in programming languages for which so-called mappings are defined. Since such mappings are available e.g. for C, C++, JAVA most business and logistic applications can be coupled via CORBA. Moreover the IDL-to-COBOL mapping allows coupling of legacy applications in these areas.

Conversely many numerical simulation codes in science and engineering are (still) written in FORTRAN for which no IDL mapping is defined. Nevertheless the growing computer power nowadays theoretically allows to run complex coupled simulation codes, for example fluid-structure interactions or climate simulations. Since the coupling of C++ and FORTRAN codes is possible, one way to couple FORTRAN codes is to use the defined IDL-to-C++ mapping and "wrap" the FORTRAN applications in C++ code.

A major issue in simulation codes is the necessary data transfer. Most simulations are time-dependent problems which are solved by a discrete time-stepping scheme. The length of the time-steps is usually determined by stability requirements and thus can not be chosen arbitrarily. To achieve real coupling between different parts of the simulation a transfer of data (e.g. boundary conditions) on the physical or problem-defined interfaces is necessary. A crucial point for efficiency thus is to avoid any data copying.

Moreover simulation codes are often run with different resolutions of the underlying mathematical equations. For this purpose flexible data structures are desirable that do not require re-compilation when the resolution changes. The IDL sequence type provides this flexibility and therefore it is a useful data type for coupled simulation codes.

For simplicity of notation in this paper a code is called a C++ code even if it does not contain any special object-oriented features and can be also compiled as pure C code. For the same reason a code is referred to as a FORTRAN code when no special FORTRAN 90 features are used. In the latter cases the notation FORTRAN 90 code will be used. Moreover all examples given for FORTRAN subroutines are also valid for functions. The files which are generated by the IDL-to-C++ translator are named according to the convention used by the ORBacus 3.1.2 software, see [2]. The compiler-specific statements refer to the IBM XL C++ and FORTRAN compiler under AIX 4.2.

Following the usual CORBA notation the component in a distributed simulation which is called by another component is denoted as the *server*, whereas the calling component in the simulation is called the *client*.

2 The basic wrapping technique

The complete definition of a mapping from IDL to a language as FORTRAN would be rather complex. On the other hand the coupling of FORTRAN and C++ codes (without CORBA) can be achieved in most cases in a rather simple and efficient way. Combining this technique with the existing IDL-to-C++ mapping thus is an appropriate way to couple FORTRAN codes via CORBA for distributed simulations.

The available Object Request Broker (ORB) software packages include an IDL translator that transforms given interface and data type definitions described in IDL into code in the corresponding language. The CORBA standard (see e.g. [3], [4]) defines in different language mappings the relations between the IDL data types and those in the corresponding language. The ORB's IDL translator then writes so-called *skeleton* code. This has to be completed by the user with an implementation of the methods defined in the interface. A simple example in IDL and C++ describing an interface for a typical simulation model with an initialization method and another one performing a single time-step is the following (e.g. defined in a file `model.idl`):

```
#include types.idl
interface model {
   void init(in p_type p);
   void step(inout x_type x);
};
```

The `in` and `inout` attributes describe whether the corresponding parameter is supposed to be changed by the method or not. It is also possible to define pure `out` parameters. The two used types are here defined in an included file. The IDL translator which is usually just by invoked by `idl model.idl` generates a skeleton code (in the files `model_skel.h` and `.cpp` in the case of the ORBacus) which defines a class `model_skel`. From this a sub-class has to be derived (e.g. in `model_impl.h`):

```
#include <model_skel.h>
class model_impl:public model_skel {
   virtual void init(const p_type& p);
   virtual void step(x_type& x);
};
```

The two methods of `model_impl` have to be implemented, either directly in C++ or by using the following so-called C++ *wrapper* which itself calls an existing FORTRAN subroutine `fstep`:

```
#include <model_impl.h>
extern "C" void fstep(...);
void model_impl::step(x_type& x) {
   fstep(...);
};
```

The `init` method can be designed as a similar wrapper for some FORTRAN `finit` subroutine. The FORTRAN subroutines have to be declared as `extern "C" void` like usual C functions. The form of the parameter list (marked here as ...) of `fstep` depends on the data type mappings from C++ to FORTRAN and some special points when passing parameters between these languages. These aspects are discussed in the next section.

3 IDL – C++ – FORTRAN mappings

The mappings from IDL to C++ are defined in the CORBA standard which introduces special C++ data types. These are implemented in the ORB software and may thus be machine-dependent. Their FORTRAN counterparts again depend on the used operating system and compilers.

For the basic data types Table 1 shows the relations or mappings between the following four levels:

- The IDL types used in the interface definitions in the `.idl` files.
- The corresponding CORBA C++ types defined by the CORBA IDL-to-C++ mapping specifications [3].
- The transformation of the latter to basic C++ types by the ORB (here for the ORBacus 3.1.2, to be found in the file `ob/include/OB/Basic.h`

in the ORBacus' installation directory). These representations depend on the sizes of basic C++ types on the current system.

- The corresponding FORTRAN types that result from the implementations by the C++ and FORTRAN compilers (here for the IBM XL compiler [5]).

It becomes clear that the last two relations have to be checked carefully according to the used machines and compilers.

Mapping general structured types requires a FORTRAN 90 compiler since derived types (as they are called in FORTRAN) are not a part of the older FORTRAN 77 standard. A structured data type defined in IDL e.g. as

```
struct {short i; float x;};
```

is mapped to C++ as

```
struct {CORBA::Short i;CORBA::Float x;};
```

which can be represented in FORTRAN as

```
type new_type
   integer(2) ::  i
   real(4) ::  x
end type new_type
```

The structure elements must have the same order and their types have to be mapped according to Table 1. One important point to note when using structures and derived types is that the C++ and FORTRAN compilers usually extend consecutive data units that together occupy less than 4 bytes (e.g. the `short` in the example above) up to the full 4 bytes. That means in the example that the following structure member `x`'s storage begins 4 bytes behind the storage of `i`. Since this is done both in C++ and FORTRAN it normally does not cause any problems. Alternatively one can force the compilers to pack the memory in the structure or derived type, respectively, by compiling the C++ code with a special option (*-qalign=packed* for the IBM XL C++ compiler), and in FORTRAN 90 by using the `sequence` attribute in the derived type definition (in a single line right after `type new_type`). Theoretically working with packed memory both in C++ and FORTRAN 90 should also work without problems, but in fact turned out not to do so in

IDL	CORBA C++ `CORBA::`	Byte	C++	FORTRAN
–	–	1	`(signed) char`	`integer(1)`
`short`	`Short`	2	`(signed) short`[2]	`integer(2)`
`long`	`Long`	4	`(signed) int`[2]	`integer(4)` =`integer`[4]
`long long`[1]	`LongLong`	8	`long long`[3]	`integer(8)`
`char`	`Char`	1	`unsigned char`	`logical(1)`
`boolean`	`Boolean`	1	`unsigned char`	`logical(1)`
`octet`	`Octet`	1	`unsigned char`	`logical(1)`
`unsigned short`	`Short`	2	`unsigned short`	`logical(2)`
`unsigned long`	`Long`	4	`unsigned int`	`logical(4)` =`logical`[4]
`unsigned long long`[1]	`LongLong`	8	`unsigned long long`[3]	`logical(8)`
`float`	`Float`	4	`float`	`real(4)`=`real`[4]
`double`	`Double`	8	`double`	`real(8)` =`double prec.`[4]
`long double`	`LongDouble`	16	`long double`[3]	`real(16)`

Table 1: Mappings for basic IDL, C++, and FORTRAN data types. Notes: [1] Type is not implemented in ORBacus. [2] In other implementations the IDL types `short` and `long` may be be mapped to the C++ types `int` and `long`, respectively. [3] Type requires special options on the IBM XLC compiler. [4] The default FORTRAN type sizes can be changed by compiler options.

some cases. Thus it is highly recommended *not* to use the memory packing. To avoid problems of this type it is moreover safe to work only with 4 or 8 byte data types as `long` in IDL, `int` in C++, and `integer(4)` in FORTRAN.

FORTRAN provides a special data type which allows to map a structure of two floating point numbers to one `complex`, e.g. the IDL type

```
struct { float re,im; };
```

which in C++ is mapped to

```
struct { CORBA::Float re,im; };
```

can be represented in FORTRAN by

`complex(4)` or just `complex`.

Analogously a `struct` of two IDL `double` can be mapped to a `complex(8)` or `double complex` and a struct of two IDL `long double` to a `complex(16)` in FORTRAN.

4 Passing parameters from C++ to FORTRAN

Besides the pure mappings of the basic data types described in the last section there are some points to take care of when passing parameters from a C++ wrapper to a FORTRAN subroutine:

- FORTRAN subroutines require their parameters to be passed by reference. That means that a subroutine

  ```
  subroutine fstep(x)
     real(4) ::  x
     ...
  end subroutine fstep
  ```

  is wrapped in C++ as

  ```
  extern "C" void fstep(CORBA::Float*);
  void model_impl::step(CORBA::Float& x) {
  ```

```
   fstep(&x);
};
```

and its IDL interface is

```
interface model {
   ...
   void step(inout float x);
};
```

Since passing a C++ array in fact means passing a pointer, the ampersand in the call of `fstep` has to be omitted in this case. The subroutine

```
subroutine fstep(x,n)
   real(4) ::  x(n)
   integer(4) ::  n
   ...
end subroutine fstep
```

is wrapped in C++ as

```
extern "C" void fstep(CORBA::FLoat*,CORBA::Long*);
void model_impl::step(FArray x,CORBA::Long n) {
   fstep(x,&n);
};
```

The introduction of a new type called `FArray` here for an array of floats is necessary to define the IDL interface in `model.idl`:

```
#define N 100
typedef float FArray[N];
interface model {
   void step(inout FArray x,in long n);
};
```

The IDL to C++ translator generates the C++ type `FArray` in the file `model.h`. Since IDL does not have a pointer type the fixed array dimension is required. For variable-sized arrays the sequence type (see below) has to be used. Instead of defining the dimension with the

`#define` preprocessor directive as above it is also possible to declare `n` itself as a constant, namely

```
const long n=100;
typedef float FArray[n];
```

But nevertheless in this case it is *not* possible to declare `n` as `extern const CORBA::Long` and use it for the definition of the `FArray` type in the C++ wrapper. In the parameter list of the prototype of `fstep` in the C++ wrapper `FArray` can be replaced by `CORBA::Float*` since this is what `FArray` in C++ in fact is.

- C++ arrays start with index 0, those in FORTRAN arrays may have arbitrary index ranges, by default they start with index 1. Since the C++ wrappers usually do not access array elements themselves this is only important to define the array's dimensions properly.

- C++ and FORTRAN have different ways when internally storing two- and more-dimensional arrays. In C++ the array elements are stored sequentially in memory with the last index running fastest, in FORTRAN with the first index doing so. Hence arrays have to be defined in the C++ wrapper and the IDL interface with the dimensions changed accordingly. The subroutine

```
subroutine fstep(x,m,n)
   real(4) ::  x(m,n)
   integer(4) ::  m,n
   ...
end subroutine fstep
```

is wrapped in C++ as

```
extern "C" void fstep(FArray,CORBA::Long*,CORBA::Long*);
void model_impl::step(FArray x,CORBA::Long n,
                      CORBA::Long m) {
   fstep(x,&m,&n);
};
```

Here the `FArray` type in the parameter list of the prototype of `fstep` is necessary if the array dimensions are not explicitly defined as constant

in the wrapper itself. The IDL interface (with `N,M` defined by preprocessor directives) is

```
typedef float FArray[N][M];
interface model {
   void step(inout FArray x,in long n,in long m);
};
```

In the array examples it becomes clear that one has either to define the array dimensions in the IDL interface sufficiently big or to translate the interface and compile the generated C++ code every time different dimensions are needed. A remedy for this unsatisfying situation provides the sequence type discussed in the last two sections of this paper.

5 Coupling FORTRAN codes without clear interfaces

Many stand-alone FORTRAN codes use global variables in common blocks or FORTRAN 90 modules to share data between different model parts that are realized in subroutines. On one hand this results in high efficiency of the code and readability of the subroutines' declarations and calls since the parameter lists remain short. On the other hand the subroutines have no clear interfaces and it thus is much more difficult to split up the software into components if parts of it shall be coupled with other software.

Nevertheless for both cases (common blocks and module variables) there is a way to wrap the code and thus provide it with a usable interface. This technique is described in the current section.

5.1 FORTRAN 77 common blocks

In the case that a FORTRAN subroutine uses common blocks to share data with other subroutines it is possible to map the common blocks to equivalent C++ structures. The common block `globals` in

```
subroutine fstep
   common / globals / x,y
   real(4) ::  x,y
```

```
   ...
end subroutine fstep
```

can be accessed in a C++ wrapper in the following way:

```
#include <model.h>
globals_type globals;
extern "C" void fstep(void);
void model_impl::step(globals_type& g_par) {
   globals=g_par;
   fstep();
   g_par=globals;
}
```

The definition of `globals_type` is included in the file `model.h` generated by the IDL-to-C++ translator from the IDL interface defined in `model.idl`:

```
struct globals_type {float x,y;};
interface model {
   void step(inout globals_type g_par);
}
```

The FORTRAN common block and the C++ structure `globals` use the same memory. Since the wrapper needs a parameter to get and pass to other CORBA components in order to transfer any data the copying from and into this parameter `g_par` is necessary. A second disadvantage of this easy coupling technique is that in the common block there may be variables that are actually not needed in the other components.

5.2 FORTRAN 90 modules

When FORTRAN 90 module variables are used they cannot be accessed via a C++ structure in the same way as a common block. Then a different method is necessary which moreover avoids the data copying. The subroutine

```
subroutine fstep
   use globals
   ...
end subroutine fstep
```

uses a global variable declared as an allocatable array in a module:

```
module globals
   real(4), dimension(:,:), allocatable ::  x
end module globals
```

An initialization subroutine allocates memory for `x`. The variable then can be accessed by all subroutines that include a `use globals` statement. Changing the `allocatable` attribute to `pointer` in the module and removing the memory allocation allows to wrap the subroutine `fstep` as follows:

```
subroutine fstep_wrapper(y,m,n)
   use globals
   real(4), target ::  y(m,n)
   integer(4) ::  m,n
   x=>y
   call fstep
end subroutine fstep_wrapper
```

The pointer assignment statement `x=>y` allows to access the values in `y` also via `x` and thus `fstep` now operates in fact on the values passed to `fstep_wrapper` in the parameter `y`. The FORTRAN 90 wrapper itself is called from C++:

```
extern "C" void fstep_wrapper(FArray,
                              CORBA::Long*,CORBA::Long*);
void model_impl::step(FArray x,
                      CORBA::Long m,CORBA::Long n) {
   fstep_wrapper(x,&m,&n);
}
```

The corresponding IDL interface is given by

```
typedef float FArray[N][M];
interface model {
   void step(inout FArray x,in long n,in long m);
}
```

Again the array is transposed in C++. Still both methods require redefinition of the IDL interfaces if the array dimensions need to be adjusted

due to a change e.g. in the resolution of the simulation's discretization. How this may be avoided is the topic of the next section.

6 Flexible interfaces using IDL sequences

Arrays whose dimensions are not fixed at compile-time but can be adjusted according to simulation requirements at run-time can be realized in FORTRAN 90 by using the `allocatable` or `pointer` attribute, in C++ by declaring a pointer to the corresponding data type. In both languages the actual needed memory then is allocated by special function or subroutine calls. Since IDL has no pointers the only way to transmit such arrays without re-parsing the interfaces and data types is the use of the sequence type. An IDL interface using the sequence type to pass an array of variable size is the following:

```
typedef sequence<float> Parameter;
interface model {
   void step(inout Parameter p,in long m,in long n);
};
```

The type `Parameter` is a so-called *unbounded sequence* where the maximal as well as the current length of the sequence can be controlled by the user. For *bounded sequences* where the maximal length is fixed in the IDL interface the following technique may be applied analogously. To "fill" the sequence without data copying a special constructor (provided in the sequence class template of the ORB implementation) is used in the C++ client that calls the wrapped FORTRAN model via CORBA:

```
x = new CORBA_Float[n*m];
Parameter p(n*m,n*m,x);
```

The sequence is built around the existing array `x` without data copying. The first two integer parameters passed to the constructor are maximal and actual length of the (unbounded) sequence. A C++ wrapper may look like this:

```
extern "C" void fstep(CORBA::Float*,
                      CORBA::Long*,CORBA::Long*);
void model::step(Parameter& p,CORBA::Long m,CORBA::Long n) {
   fstep(p.data(),&m,&n);
```

```
};
```

The `data` function of the `Parameter` class is an additional function supplied by the ORBacus. It returns the pointer to the data array of the sequence type object `p`. The FORTRAN subroutine

```
subroutine fstep(x,m,n)
   real(4) ::  x(m,n)
   integer(4) ::  m,n
   ...
end subroutine fstep
```

or the subroutine `fstep_wrapper` defined at the end of the last section can be used here even though they interpret `x` as the reference to an array of dimension greater than one. Care has to be taken for the way the one-dimensional array `x` is filled on the client side: The FORTRAN subroutine regards the passed `x` as reference to a (in this case) two-dimensional array of size `m`×`n` which was filled with the data *column-wise*.

Note that this method only works for arrays of basic data types and not for arrays of structures or derived types. In these cases the C++ pointer returned by `p.data()` cannot be interpreted correctly by the called FORTRAN subroutine. To deal with this case the whole sequence has to be passed to `fstep` in the way discussed in the next section.

7 A FORTRAN mapping for IDL sequences

In the last section the sequence type was used to define an interface that transfers allocatable arrays of floating point numbers. The sequence was resolved in the C++ wrapper and then the address of the array itself was passed to FORTRAN.

A more general and flexible method is to define a mapping of the IDL sequence type or its C++ representation. Then the sequence may be passed unchanged through the C++ wrapper directly to a FORTRAN wrapper or subroutine. This may be necessary if a bigger number of arrays have to been passed to the subroutine and the number and the dimensions of the arrays should be variable. Then a sequence of sequences may serve as a flexible and efficient data type for a parameter list. Sequences are also necessary in cases when arrays of structures shall be transmitted. In this section a more

complicated example is presented where a structure contains pointers to an array of other structures.

To construct a derived type in FORTRAN 90 that may serve as a representation of a sequence it is necessary to know how a sequence is realized in C++, namely as a class whose attributes are integers for the maximal and actual length, an optional offset, a storage release flag, and finally a pointer to the first element of the sequence itself, realized in the class template:

```
template<class T> class OBVarSeq {
   CORBA::Long max_;
   CORBA::Long len_;
   CORBA::Long off_;
   CORBA::Boolean rel_;
   T* data_;
   ...
}
```

The remaining part contains different constructors, e.g. the one already used in the last section, and other functions and operators.

A problem now arises in finding an appropriate counterpart of the `T* data` pointer in this definition. The FORTRAN 90 pointer data type differs from the C++ pointer in the way that it contains not just only the pure address (i.e. an integer) of the object the pointer points to (for simplicity called *pointee* here). FORTRAN 90 pointers moreover include information about the pointee, e.g. its rank, dimensions, stride etc. (for an array). The problem is that the way how this information is stored is not standardized and usually not documented by the compiler vendors. Passing a C++ pointer pointing to a struct as parameter to a FORTRAN 90 subroutine which tries to interpret it as a pointer to a derived type leads to run-time errors, even if the FORTRAN 90 derived type has exactly the same elements in the same order and with the appropriate types as its C++ counterpart. A way to overcome these problems is the use of the so-called *integer* or *Cray pointers*. They are not part of the FORTRAN 77/90/95 standards but nevertheless supported by most Fortran compilers. The *integer pointer* is an integer that can be associated with a pointee by a `pointer` statement. Since the pointee can be of any type the same derived type can be used for IDL sequences of arbitrary type. For convenience it is defined in a FORTRAN 90 module:

```
module seq_mod
```

```
  type seq
    private
    integer(4) ::  max,len,off
    logical(4) ::  rel
    integer(4) ::  dat
  end type seq
contains
  integer(4) function length(s)
    type(seq) s
    length = s%len
    return
  end function length
  integer(4) function dat(s)
    type(seq) s
    dat=s%dat
    return
  end function dat
end module seq_mod
```

The C++ `CORBA::Boolean rel` can be mapped either to a FORTRAN `logical(1),(2)` or `(4)` for the reasons already discussed at the end of Section 3. In the first two cases the attribute `sequence` *must not appear* in the FORTRAN 90 derived type definition, whereas in the last it has no effect. The module does not allow direct access to the types' elements, but it provides two functions returning the length and the integer pointer to the data in the sequence, respectively. The second function thus corresponds to the `data` member function in the class definition of the C++ mapping of the IDL sequence type. (`data` is a keyword in FORTRAN – it can be used as name for this function but is avoided here.)

The module may now be included anywhere a sequence is passed to a FORTRAN subroutine. The following example shows how a C++ structure originally containing a pointer to an array of other structures is transformed to a sequence and transmitted to a FORTRAN subroutine. Based on the C++ type definition

```
struct Parameter {float x,y;};
struct Transfer {int n;float z;Parameter *list;};
```

an object of type `Transfer` containing a list with an arbitrary number of

parameters shall be passed to FORTRAN. In the IDL interface defined in `model.idl` the pointer is replaced by a sequence object:

```
struct Parameter {float x,y;};
typedef sequence <Parameter> ParList;
struct Transfer {long n; float z; ParList list;};
interface model {
   void step(inout Transfer pt);
};
```

On the server side the following C++ wrapper is used:

```
extern "C" void fstep(Transfer*);
void model_impl::step(Transfer& p) {
   fstep(&p);
}
```

The IDL and the corresponding C++ types are mapped to FORTRAN 90 derived types in a module which uses the `seq_mod` module:

```
module transfer_mod
  use seq_mod
  type Par
    real(4) ::  x,y
  end type Par
  type Transfer
    integer (4) ::  n
    real(4) ::  z
    type(seq) ::  list
  end type Transfer
end module transfer_mod
```

The FORTRAN subroutine receiving a reference to an object of type `transfer` accesses the sequence using an integer pointer in the following way:

```
subroutine fstep(pt)
  use transfer_mod
  type(Transfer) ::  pt
  type(Par) ::  p(pt%n)
```

```
    pointer(ptr,p)
    ptr=dat(pt%list)
    p(1)%x=...
  end subroutine fstep
```

The `pointer` statement associates the integer pointer `ptr` with the pointee `p`. Any object of type `Par` to which `ptr` will point later on can be referred to as `p`. Now `ptr` can be assigned to any other integer pointer. This is done here using the `dat` function supplied by the `seq_mod` module.

Summary

Using the given IDL-to-C++ mapping and taking care of the relations between C++ and FORTRAN data types FORTRAN codes can be coupled via CORBA and C++ wrappers. This is even true for codes using global variables for the data transfer and without explicit interfaces. The IDL sequence type provides a flexible data type to transfer arrays and structures whose length is determined only at run-time. Even for complex structures including pointers the sequence type can be used by defining a special FORTRAN 90 derived type.

Acknowledgements

The author would like to thank Arnulf Günther and Cezar Ionescu from PIK for their comments on the manuscript and other fruitful discussions.

References

[1] The Object Management Group (OMG) homepage, http://www.omg.org/gettingstarted/corbafaq.htm.

[2] Object Oriented Concepts (OOC) homepage, http://www.ooc.com.

[3] The Object Management Group, C++ mapping specification, http://www.omg.org/technology/documents/formal/c++.htm.

[4] S. Henning, M. Vinoski, Advanced CORBA Programming with C++ (Addison-Wesley, Reading, 1999).

[5] IBM XL Fortran User's Guide - Passing data between languages, available online (command `info -l xlf`).

PIK Report-Reference:

No. 1 3. Deutsche Klimatagung, Potsdam 11.-14. April 1994,
Tagungsband der Vorträge und Poster (April 1994)

No. 2 Extremer Nordsommer '92
Meteorologische Ausprägung, Wirkungen auf naturnahe und vom Menschen beeinflußte Ökosysteme, gesellschaftliche Perzeption und situationsbezogene politisch-administrative bzw. individuelle Maßnahmen (Vol. 1 - Vol. 4)
H.-J. Schellnhuber, W. Enke, M. Flechsig (Mai 1994)

No. 3 Using Plant Functional Types in a Global Vegetation Model
W. Cramer (September 1994)

No. 4 Interannual variability of Central European climate parameters and their relation to the large-scale circulation
P. C. Werner (Oktober 1994)

No. 5 Coupling Global Models of Vegetation Structure and Ecosystem Processes - An Example from Arctic and Boreal Ecosystems
M. Plöchl, W. Cramer (Oktober 1994)

No. 6 The use of a European forest model in North America: A study of ecosystem response to climate gradients
H. Bugmann, A. Solomon (Mai 1995)

No. 7 A comparison of forest gap models: Model structure and behaviour
H. Bugmann, Y. Xiaodong, M. T. Sykes, Ph. Martin, M. Lindner, P. V. Desanker, S. G. Cumming (Mai 1995)

No. 8 Simulating forest dynamics in complex topography using gridded climatic data
H. Bugmann, A. Fischlin (Mai 1995)

No. 9 Application of two forest succession models at sites in Northeast Germany
P. Lasch, M. Lindner (Juni 1995)

No. 10 Application of a forest succession model to a continentality gradient through Central Europe
M. Lindner, P. Lasch, W. Cramer (Juni 1995)

No. 11 Possible Impacts of global warming on tundra and boreal forest ecosystems - Comparison of some biogeochemical models
M. Plöchl, W. Cramer (Juni 1995)

No. 12 Wirkung von Klimaveränderungen auf Waldökosysteme
P. Lasch, M. Lindner (August 1995)

No. 13 MOSES - Modellierung und Simulation ökologischer Systeme - Eine Sprachbeschreibung mit Anwendungsbeispielen
V. Wenzel, M. Kücken, M. Flechsig (Dezember 1995)

No. 14 TOYS - Materials to the Brandenburg biosphere model / GAIA
Part 1 - Simple models of the "Climate + Biosphere" system
Yu. Svirezhev (ed.), A. Block, W. v. Bloh, V. Brovkin, A. Ganopolski, V. Petoukhov, V. Razzhevaikin (Januar 1996)

No. 15 Änderung von Hochwassercharakteristiken im Zusammenhang mit Klimaänderungen - Stand der Forschung
A. Bronstert (April 1996)

No. 16 Entwicklung eines Instruments zur Unterstützung der klimapolitischen Entscheidungsfindung
M. Leimbach (Mai 1996)

No. 17 Hochwasser in Deutschland unter Aspekten globaler Veränderungen - Bericht über das DFG-Rundgespräch am 9. Oktober 1995 in Potsdam
A. Bronstert (ed.) (Juni 1996)

No. 18 Integrated modelling of hydrology and water quality in mesoscale watersheds
V. Krysanova, D.-I. Müller-Wohlfeil, A. Becker (Juli 1996)

No. 19 Identification of vulnerable subregions in the Elbe drainage basin under global change impact
V. Krysanova, D.-I. Müller-Wohlfeil, W. Cramer, A. Becker (Juli 1996)

No. 20 Simulation of soil moisture patterns using a topography-based model at different scales
D.-I. Müller-Wohlfeil, W. Lahmer, W. Cramer, V. Krysanova (Juli 1996)

No. 21 International relations and global climate change
D. Sprinz, U. Luterbacher (1st ed. July, 2n ed. December 1996)

No. 22 Modelling the possible impact of climate change on broad-scale vegetation structure - examples from Northern Europe
W. Cramer (August 1996)

No. 23 A methode to estimate the statistical security for cluster separation
F.-W. Gerstengarbe, P.C. Werner (Oktober 1996)

No. 24 Improving the behaviour of forest gap models along drought gradients
H. Bugmann, W. Cramer (Januar 1997)

No. 25 The development of climate scenarios
P.C. Werner, F.-W. Gerstengarbe (Januar 1997)

No. 26 On the Influence of Southern Hemisphere Winds on North Atlantic Deep Water Flow
S. Rahmstorf, M. H. England (Januar 1977)

No. 27 Integrated systems analysis at PIK: A brief epistemology
A. Bronstert, V. Brovkin, M. Krol, M. Lüdeke, G. Petschel-Held, Yu. Svirezhev, V. Wenzel (März 1997)

No. 28 Implementing carbon mitigation measures in the forestry sector - A review
M. Lindner (Mai 1997)

No. 29 Implementation of a Parallel Version of a Regional Climate Model
M. Kücken, U. Schättler (Oktober 1997)

No. 30 Comparing global models of terrestrial net primary productivity (NPP): Overview and key results
W. Cramer, D. W. Kicklighter, A. Bondeau, B. Moore III, G. Churkina, A. Ruimy, A. Schloss, participants of "Potsdam '95" (Oktober 1997)

No. 31 Comparing global models of terrestrial net primary productivity (NPP): Analysis of the seasonal behaviour of NPP, LAI, FPAR along climatic gradients across ecotones
A. Bondeau, J. Kaduk, D. W. Kicklighter, participants of "Potsdam '95" (Oktober 1997)

No. 32 Evaluation of the physiologically-based forest growth model FORSANA
R. Grote, M. Erhard, F. Suckow (November 1997)

No. 33 Modelling the Global Carbon Cycle for the Past and Future Evolution of the Earth System
S. Franck, K. Kossacki, Ch. Bounama (Dezember 1997)

No. 34 Simulation of the global bio-geophysical interactions during the Last Glacial Maximum
C. Kubatzki, M. Claussen (Januar 1998)

No. 35 CLIMBER-2: A climate system model of intermediate complexity. Part I: Model description and performance for present climate
V. Petoukhov, A. Ganopolski, V. Brovkin, M. Claussen, A. Eliseev, C. Kubatzki, S. Rahmstorf (Februar 1998)

No. 36 Geocybernetics: Controlling a rather complex dynamical system under uncertainty
H.-J. Schellnhuber, J. Kropp (Februar 1998)

No. 37 Untersuchung der Auswirkungen erhöhter atmosphärischer CO_2-Konzentrationen auf Weizenbestände des Free-Air Carbondioxid Enrichment (FACE) - Experimentes Maricopa (USA)
Th. Kartschall, S. Grossman, P. Michaelis, F. Wechsung, J. Gräfe, K. Waloszczyk, G. Wechsung, E. Blum, M. Blum (Februar 1998)

No. 38 Die Berücksichtigung natürlicher Störungen in der Vegetationsdynamik verschiedener Klimagebiete
K. Thonicke (Februar 1998)

No. 39 Decadal Variability of the Thermohaline Ocean Circulation
S. Rahmstorf (März 1998)

No. 40 SANA-Project results and PIK contributions
K. Bellmann, M. Erhard, M. Flechsig, R. Grote, F. Suckow (März 1998)

No. 41 Umwelt und Sicherheit: Die Rolle von Umweltschwellenwerten in der empirisch-quantitativen Modellierung
D. F. Sprinz (März 1998)

No. 42 Reversing Course: Germany's Response to the Challenge of Transboundary Air Pollution
D. F. Sprinz, A. Wahl (März 1998)

No. 43 Modellierung des Wasser- und Stofftransportes in großen Einzugsgebieten. Zusammenstellung der Beiträge des Workshops am 15. Dezember 1997 in Potsdam
A. Bronstert, V. Krysanova, A. Schröder, A. Becker, H.-R. Bork (eds.) (April 1998)

No. 44 Capabilities and Limitations of Physically Based Hydrological Modelling on the Hillslope Scale
A. Bronstert (April 1998)

No. 45 Sensitivity Analysis of a Forest Gap Model Concerning Current and Future Climate Variability
P. Lasch, F. Suckow, G. Bürger, M. Lindner (Juli 1998)

No. 46 Wirkung von Klimaveränderungen in mitteleuropäischen Wirtschaftswäldern
M. Lindner (Juli 1998)

No. 47 SPRINT-S: A Parallelization Tool for Experiments with Simulation Models
M. Flechsig (Juli 1998)

No. 48 The Odra/Oder Flood in Summer 1997: Proceedings of the European Expert Meeting in Potsdam, 18 May 1998
A. Bronstert, A. Ghazi, J. Hladny, Z. Kundzewicz, L. Menzel (eds.) (September 1998)

No. 49 Struktur, Aufbau und statistische Programmbibliothek der meteorologischen Datenbank am Potsdam-Institut für Klimafolgenforschung
H. Österle, J. Glauer, M. Denhard (Januar 1999)

No. 50 The complete non-hierarchical cluster analysis
F.-W. Gerstengarbe, P. C. Werner (Januar 1999)

No. 51 Struktur der Amplitudengleichung des Klimas
A. Hauschild (April 1999)

No. 52 Measuring the Effectiveness of International Environmental Regimes
C. Helm, D. F. Sprinz (Mai 1999)

No. 53 Untersuchung der Auswirkungen erhöhter atmosphärischer CO_2-Konzentrationen innerhalb des Free-Air Carbon Dioxide Enrichment-Experimentes: Ableitung allgemeiner Modellösungen
Th. Kartschall, J. Gräfe, P. Michaelis, K. Waloszczyk, S. Grossman-Clarke (Juni 1999)

No. 54 Flächenhafte Modellierung der Evapotranspiration mit TRAIN
L. Menzel (August 1999)

No. 55 Dry atmosphere asymptotics
N. Botta, R. Klein, A. Almgren (September 1999)

No. 56 Wachstum von Kiefern-Ökosystemen in Abhängigkeit von Klima und Stoffeintrag - Eine regionale Fallstudie auf Landschaftsebene
M. Erhard (Dezember 1999)

No. 57 Response of a River Catchment to Climatic Change: Application of Expanded Downscaling to Northern Germany
D.-I. Müller-Wohlfeil, G. Bürger, W. Lahmer (Januar 2000)

No. 58 Der "Index of Sustainable Economic Welfare" und die Neuen Bundesländer in der Übergangsphase
V. Wenzel, N. Herrmann (Februar 2000)

No. 59 Weather Impacts on Natural, Social and Economic Systems (WISE, ENV4-CT97-0448) German report
M. Flechsig, K. Gerlinger, N. Herrmann, R. J. T. Klein, M. Schneider, H. Sterr, H.-J. Schellnhuber (Mai 2000)

No. 60 The Need for De-Aliasing in a Chebyshev Pseudo-Spectral Method
M. Uhlmann (Juni 2000)

No. 61 National and Regional Climate Change Impact Assessments in the Forestry Sector - Workshop Summary and Abstracts of Oral and Poster Presentations
M. Lindner (ed.) (Juli 2000)

No. 62 Bewertung ausgewählter Waldfunktionen unter Klimaänderung in Brandenburg
A. Wenzel (August 2000)

No. 63 Eine Methode zur Validierung von Klimamodellen für die Klimawirkungsforschung hinsichtlich der Wiedergabe extremer Ereignisse
U. Böhm (September 2000)

No. 64 Die Wirkung von erhöhten atmosphärischen CO_2-Konzentrationen auf die Transpiration eines Weizenbestandes unter Berücksichtigung von Wasser- und Stickstofflimitierung
S. Grossman-Clarke (September 2000)

No. 65 European Conference on Advances in Flood Research, Proceedings, (Vol. 1 - Vol. 2)
A. Bronstert, Ch. Bismuth, L. Menzel (eds.) (November 2000)

No. 66 The Rising Tide of Green Unilateralism in World Trade Law - Options for Reconciling the Emerging North-South Conflict
F. Biermann (Dezember 2000)

No. 67 Coupling Distributed Fortran Applications Using C++ Wrappers and the CORBA Sequence Type
Th. Slawig (Dezember 2000)