CTL Manual for Linux/Unix for the Usage with C++

Contents

1 Introduction

The Component Template Library (CTL) is an implementation of the component technology based on C++ generic template programing \cite{tplcpp}. Similar to CORBA \cite{CORBA} it can be used to realize distributed component-based software systems, where a component is a piece of software which consists of a well defined interface and an implementation. Interface and implementation are connected throw a communication channel, e.g. TCP/IP or MPI. In this way the usage of a component is independend of their location, meaning that the component is network transparent.

One important feature of the CTL is their lightweight design and their seamless integration into the C++ language. It is a template library and in its simplest configuration it depends on nothing than the standard libraries which are available on all Unix plattforms. Like the standard template library (STL) — which is a part of the C++ language — provides generic container datatypes, the CTL provides methods to realize distributed systems.

The CTL can be used for fast prototyping of distributed software systems. But its main focus is to transform existing C/C++ or FORTRAN libraries to remote accessable software components. The idea behind the CTL is to provide a mechanism which makes the development of distributed systems as easy as possible, so that the differences between traditional monolithic programs and complex distributed software systems nearly vanish.

Main design aspects for the CTL are:

- easy syntax (using overloading of suizable operators)
- header only (no precompile or install)
- independance of other libraries (needs only sockets, dlopen, pthreads,
- mpi/pvm can be used optionally)
- expandable for other communication protocols (open communication interface and protocol)
- maximal decoupling of components (using idl, abstract types)
- covering the parallel and the distributed programming models (group and links)
- direct process to process communication (no daemon in between)
- uniform behaviour of remote (tcp/ip, mpi, pvm, pipes, daemons) and local (library, thread) linkage types
- extrinsic usage of user defined types
- support of user defined types

The main idea (as Com, Corba, ...) is to define an interface description of a library that gives all informations two components must share in order to understand each other in a type safe way. This information includes the names and signatures of functions, classes and their methods. In this context the signature of a function, method, static method or constructor specifies its name, the argument list, the type of the result as well as the types of exceptions which might be thrown.

2 first examples

Any dual interactions have three parts: the participating components and the coupling between them. Therefore the examples have to handle the component interface, the caller and the callee side .In the following CI is used as short cut for Component Interface.

2.1 classes

A CI interface may contain class definitions.

2.1.1 Interface

A CI class definition might look like the following example.

```
#ifndef _1ST_CI_
#define _1ST_CI_

#include ctl.h

#define CTL_Class firstClass
#include CTL_ClassBegin

#  define CTL_Constructor1 (const string /*init-
file*/), 1

#  de-
fine CTL_Method1 int4, f, (const real8) const, 1
#  de-
fine CTL_Method2 void, g, (real8, const bool), 2

#  define CTL_StaticMethod1 bool, s, (), 0

#include CTL_ClassEnd

#endif
```

The C++ notation of this class is

```
class firstClass
{
public:
```

```
    firstClass(const string /*initfile*/&);

    int4 f(const real8&) const;
    void g(real8, const bool);

    static bool s();
  };
```

The CTL interface description is defined in terms of C-macro definitions which are expanded to the functions which perform the data serialisation and transport. Due to the fact, that the C-preprocessor can not count lists, at the end of each declaration the number of arguments must be given.

2.1.2 Caller

```
#include firstCI.h

int main()
{
  first-
Class::use(”host:/usr/peter/comp/libsecondImpl.exe tcp”);

  firstClass C(”initfile”);

  int n=3;
  double x=3.3;
  std::cout<<C.f(x)<<std::endl;
  C.g(x, true);
  std::cout<<x<<std::endl;
  std::cout<<firstClass::s()<<std::endl;
}
```

2.1.3 Callee

```
#define CTL_Connect
#include ”firstCI.h”
#include <string>

class firstImpl
{
public:
  firstImpl(const std::string &fileName)
  { std::cout<<filename<<std::endl; }

  int f(double x) const
```

```
  { return int(x)+1;}

  void g(double &x, bool sw)
  {
    if(sw)
      x = 0;
  }
  static bool s()
  { return true; }
};

void CTL_connect()
{
  ctl::connect<firstClass, firstImpl>();
}
```

2.2 Libraries

A CI library may contain CI class definitions and CI function declarations.

2.2.1 Interface

```
#ifndef _2nd_CI_
#define _2nd_CI_

#include ctl.h

#define CTL_Library secondCI
#include CTL_LibBegin

#  define CTL_Function1 int4, f, (const real8), 1
#  de-
fine CTL_Function2 void, g, (real8, const bool), 2

#include CTL_LibEnd

#endif
```

2.2.2 Caller

```
#include secondCI.h

int main()
{
  sec-
ondCI::use(”/usr/peter/comp/secondImpl.so thread”);
```

```
  int n=3;
  double x=3.3;
  std::cout<<secondCI::f(x)<<std::endl;
  secondCI::g(x, true);
  std::cout<<x<<std::endl;
}
```

2.2.3 Callee

```
#define CTL_Connect
#include ”secondCI.h”

int4 f(real8 x)
{ return int(x)+1;}

void g(real8 &x, bool sw)
{ if(sw)
    x = 0;
}

void CTL_connect()
{
  secondCI::connect(&secondCI::f, &f);
  secondCI::connect(&secondCI::g, &g);
}
```

2.3 basic types

2.3.1 ctl::location

```
class location
{
  void setTerminal (const std::string &trm);
//assign a terminal.

  void setDebug (const std::string &dbg);
//assign a debugger.

  location (const char *locC, linkage =unde-
fined);
  location (const std::string &locStr, link-
age =undefined)
//Constructor accepting a string in ssh syntax.

  loca-
```

```
tion (const std::string &e, const std::string &p, const std::string &h=std::
age =tcp)
//constructor assigning the exe-
cutable, the path, and   optionally the host (de-
fault localhost) and the     linkage (de-
fault tcp).
};
```

modifiers for location:

- l <linkage> in {lib,thread, tcp, pipe, lam, mpi, pwm, dmn,...}
- f <logfile> (absolut or relative to current working directory of destination component)
- d <working directory> (absolut or relative to executable)
-v be verbose
-p logical id in group (default=-1)
-S size of group (default=0)
-h hostname/IP of client
-x <terminal> run in terminal (default = xterm -e)
-g <debugger> start in debugger, implies -x (default = gdb)
-s <remote shell> at <port> (defaults = ssh at 22)
-n <value> set priority (default value = 1) decrement process priority by value
-L <port> use as local port
-R <port> use as remote port
-c cypher messages

2.3.2 ctl::link

```
class link
{
  link ()
// no ressource allocation.

  link (const ctl::any &property)
//Takes a property for the location selector.

  link (const std::string &loc, link-
age typ=undefined)
//if the linkage is file loc is inter-
preted as an filename, otherwise as a loca-
tion in ssh syntax.

  link (const location &loc, link-
age typ=undefined)
//create a link using the given loca-
tion and the specified linkage.
```

```
  bool operator! () const
// check wether the link is still valid.

  int test() const;
// return size of data available, -1 otherwise
  template<class T> const link &operator<<(const T&) const;
  template<class T> const link &operator>>(T&) const;
// send/recv object of type T
};
```

2.3.3 ctl::group

```
class group
{
  template<class LocVec> group (const LocVec &, linkage)
// create a group by the sample of location ’s and the given linkage.

  group (int n, const char *const *arg, const char *logFile=0, bool cwd=false, bool enableDmn=false)
// create a group by the command line arguments, logFile if given will define the the log file(s), if cwd is set the working dir is set corresponding to arg[0].

   bool operator! () const;
// check whether the group is still valid.

    bool run () const
// blocking:run a distributed application until termination is detected.

  link operator[] (int p) const
// return the specified link of this group.

  int size () const
// return the size of this group.

  int logId () const
// returns the logical Id of this instance of the group.
```

```
   template<class T> T global-
Sum (const T &t) const;
   template<class T> T global-
Prod (const T &t) const;
   template<class T> T glob-
alMin (const T &t) const;
   template<class T> T global-
Max (const T &t) const;
// blocking: com-
pute the sum, prod, min, max of t over all pro-
cesses in this group.
};
```

2.3.4 ctl::result<Y>

```
template<class Y> class result
{
  result (const result< Y > &)
// implements the owner concept as std::auto_ptr.

  result<Y>& operator= (const result< Y > &)
// implements the owner con-
cept as the std::auto_ptr.

  bool operator! () const
// check whether result is available.

  operator const Y& () const
// blocking: return the returned value.
// update all non const arguments of former call
};
```

2.3.5 global functions

set the Locator for implicit loaction selection

```
template<class locSlct>
  void ctl::setLocator(const locSlct&);
```

enable polymorphic IO of Type

```
ctl::connect<Type>();

// user defined
void CTL_connect();
```

2.3.6 CI functions

A library or class interface has together with the function declared in the interface the following static functions:

```
CI::connect<Impl, Detail = void>();

CLib::connect<CIfunc, Implfunc>(CIfunc*, Impl-
func*);
CLib::connectID<FID, Implfunc>(Implfunc*);

ctl::link &CLib::use(const ctl::link & =ctl::link());
ctl::link &CI::use(const ctl::link & =ctl::link());
```

A CI class has additionally to constructors declared in the interface the template constructor

```
template<class Impl> CI::CI(const Impl &);
```

where `Impl` must be a valid implementation of CI, that meens `Impl` defines all constructors, methods and static methods declared in the interface in a way that the overloading rules given in section <overloading> are sucessfully applied. So a CI class can be implemented locally and acts therefor here as an extrinsic interface class (a g++ class signature).

2.4 nonblocking calls

example: receive result in blocking way

```
// blocking call
int n = firstCI::f(x);
// non blocking call
```

receive result in non blocking way

```
ctl::result<int4> res = firstCI::f(x);
while(!res)
{
// do other stuff
}
int n = res;
```

receive a couple of results in non blocking way

```
vector<double> task;
vector<int> result(task.size());
vector<firstCI> worker(nWorker);
// instantiate worker
vector<int> state(worker.size(), -1);
vector<ctl::result<int> > resHan-
dle(worker.size());
int read = 0, curr = 0;
while(read<task.size())
{
  for(int p=0; p<worker.size(); p++)
  {
    if(state[p]>=0 && !!resHandle[p])
    {
      read++;
      result[state[p] ] = resHandle[p];
      state[p] = -1;
    }
    if(curr<task.size() && state[p]<0)
    {
      state[p] = curr;
      resHandle[p] = worker[p].f(task[curr++]);
    )
  }
}
```

2.5 template classes

2.5.1 Interface

```
# define CTL_ClassTmpl sorterRI, (S,T), 2
# include CTL_ClassBegin
#  define CTL_StaticMethod void, sort, (ar-
ray<S>), 1
#include CTL_ClassEnd
```

defines a class template with the type parameters S and T as

```
template<class S, class T> class sorterCI
```

Inside the class body S and T are valid type identifier.

2.5.2 caller

```
sorterCI<double, int> sorter(link);
std::list<double> val;
sorter.sort(val);
```

2.5.3 callee

```
ctl::connect<sorterCI<double, int>,myDoubleSort>();
ctl::connect<sorterCI<int, char>, myIntSort>();
```

2.6 template functions

The declaration corresponds to

```
# de-
fine CTL_FunctionTmpl1 void, (h, (S,T), 2), (const S ,T), 2
```

corresponds to

```
template<class S, class T> void h(const S&, T&);
```

2.7 connecting non matching classes

In the case an CI class has to be implemented by an existing library more often the method names will not match.

In another case explicite overload resolution might be necassary. For this purposes one may define a structure with the meaning of a connection detail and give it as a type parameter to the class connector.

2.7.1 bind constructors

```
CTL_Constructor(cid,(args),#args)
```

2.7.2 bind methods

```
CTL_Method(mid, y, meth,(args),#args)
```

2.7.3 bind static methods

```
CTL_StaticMethod(fid, y, func,(args),#args)
```

Example:

```
struct connectDetail
{
  CTL_Constructor(1, (const std::string &), 1);

  CTL_StaticMethod(1, int, Parti-
cleImpl::h,   (std::vector<double>&, dou-
ble), 2);
```

```
  CTL_Method(2, float, Parti-
cleImpl::f, (std::vector<int>, float), 2);
};
void CTL_connect()
{ // connect classes
    ctl::connect<TracerCI,ParticleImpl,connectDetail>();
}
```

Remarks:

1. The name of connect detail class is user defined and therefore arbitrary:
2. The connect detail parameter has only effect to the constructors, static and non static methods explicitly given in that connect detail
3. A static method can be connected to an arbitrary global function (this includes public static methods of arbitrary classes)

2.8 overloading resolution

The following overloading resolution rules are applied in the given sequence.

- If a constructor, method, or static method is connected explicitly, the connected one will be called (a gobal function or function template must be connected explicitly via CI::connect()).
- If the argument list contains no abstract type, the usual C++ overloading rules are valid.
- If the argument list contains at least one abstract type, the method or static method of the implementation class with the name given in the interface is chosen. Overloading is in this case not possible.

These rules imply that a constructor with an abstract argument must be connected explicitly using CTL_Constructor.

2.9 Exceptions

Any CI-method, constructor or function may throw an exception. In order to catch such an exception on the client side the types of exceptions which might be thrown must be given in te Interface-Declaration.

Example

```
#define CTL_Class firstClass
...
#  de-
fine CTL_Method1 int4, f, (const real8) const, 1
```

```
# define CTL_Method1Throws (std::string, std::exception),2
...
```

client:

```
firstClass c1;
try {
  int i = c1.f(3.1416);
}
catch(const std::string &)
{ ... }
catch(const std::exception &)
{ ... }
catch(...)
{ ... }
```

or

```
ctl::result<int> r=c1.f(3.1416);// nothing to catch here
while(!r) { // nothing to catch here
//  do other stuff
}
try{ // catch exception where the result is casted
  int i=r;
}
catch(const std::string &)
{ ... }
catch(const ctl::exception &)
{ ... }
catch(...)
{ ... }
```

3 portable data types

Datatypes which appear in CTL interfaces must be transportable over a network. This section explains the basic datatypes which are already prepared to work with the CTL and describes how to serialize user defined datatypes.

3.1 standard types

3.2 stl types

Instantiations of the stl template types:

vector<T, allocator>, list<T, allocator>, set<T, compare, allocator>,
map<key, value, compare, allocator>,
basic_string<T, allocator> are portable
with portable T, key and value are portable if T, key and value are portable.

3.3 Type Naming

In order to serialze types which are polymorph, some type identification is needed, see sections <any> , <polymorph io> and <references> Herefor the typeinfo.name() function could be used if it would give the same name for each C++-Compiler, what is not the case. Therefor the CTL introduces it's own type naming

3.3.1 intrinsic

These macros can be used inside a class/ template class definitions.

```
CTL_TypeName(T)
CTL_TemplateName(T, (X1, X2,...,Xn), n)
```

3.3.2 extrinsic

These macros can be used in the global namespace.

```
CTL_SetTypeName(T)
CTL_SetTemplateName(T, (X1, X2,...,Xn), n)
```

Examples:

```
namespace wire
{
  class AlgorithmBase
  {
   public:
    virtual CTL_TypeName(wire::AlgorithmBase);
  };
  template<class T> class Algorithm : pub-
lic     AlgorithmBase
  {
   public:
    CTL_TemplateName(wire::Algorithm, (T), 1);
  };
} // wire
// or alternativly without virtual call mechanism
  CTL_SetTemplateName(wire::Algorithm, (T), 1);
```

Assigns AlgorithmBase the type name “wire::AlgorithmBase” and Algorithm<char*> the name “wire::Algorithm<char*>”, where the typename of AlgorithmBase will be extracted using the virtual method call mechanism. In the following we say a type is named iff there is a CTL name definition for this type. All standard types and the types listed in section <stl::types> are named.

.

3.4 the type ctl::any

In order to implement generic algorithms working with objects of arbitrary non uniform types an implementation of an any type is useful. The type any has a constructor excepting an object x of arbitrary type X. The conversion of an any to a Y* gives a valid pointer of type Y iff Y is X or derived by X. The type any implements a value semantic based on the late copy mechanism. The any object is successfull transportable iff X is transportable, named and connected.

If one link reads the any object where X is not named and connected or X is even not defined the readed any object will carry only the binary data representation of x. It may be send to further processes, which iff X is there transportable, named and connected may sucessfully perform the recast to X*.

```
class any
{
// construction
  any();
  template<class T> any (const T &t);
  any(const any &obj);
// assignment
  template<class T> any &operator= (const T &t);
  any &operator = (const any &obj);
  template<class T> operator T*();
// conversion
  template<class T> operator const T*() const;
  void clear();
  bool operator !() const;
};
```

3.5 user defined types

A datatype or class which might be used in a CTL interface as an argument or return type must be transportable. The basic datatypes which comes with the CTL (see section \ref{sec::datatypes}) as well as the basic stl container types are already prepared to be transportable. Transportable does mean, that the object can be serialized into a datastream in order to send it over a communication channel. If one wants to use user defined datatypes or classes inside methods or functions of a CTL interface, one have to serialize the class. Therefore the CTL provides a macro called `CTL_Type`.

```
namespace wire
{
```

```
  class Data
  { public:
    std::string _host;
    std::string _path;
    std::string _service;
    std::string _user;
    std::string _password;
    int _count;
    double _time;
    Data(){}
    ~Data(){}
    CTL_Type(wire::Data, tu-
pel,(_host,_path,_service,_user,_password,_count,_time), 7)
  }

  template<class T> class Vector
  {
    void resize(int);
    T *begin();
    T *end();
    CTL_Template(wire::Vector, array, (be-
gin, end, size, resize),4, (T), 1)
  };
};
```

3.5.1 intrinsic definition of serialization

Put into the (template) class body a

```
CTL_Type/CTL_Template
```

statement or write

```
ctl::oStream &write(ctl::oStream &os) const
{ ... }
ctl::iStream &read(ctl::iStream &is)
{ ... }
```

Remarks:

It’s not a good idea to have virtual read/write methods, because the consistancy of the interpretation of the binary stream between writer and reader will most likely be destroyed, see Section <polymorph types> how to realize polymorph IO.

3.5.2 extrinsic definition of serialization

Put into the global namespace:
CTL_SetType/CTL_SetTemplate
or write

```
namespace ctl
{
  oStream &write(oStream &os,const T &t)
  {
     // write attributes of T in some order
      ...
  }
  iStream &read(iStream &is,T &t)
  {
    // read attributes of T in the same!! order
     ...
  }
}
```

Remark: Beware carefully the consistancy/symmetry of the write and read functions otherwise the stream interpreation at the reading side will be corrupted and system behavior becomes undefined.

The expansions of the macros CTL_Type/CTL_Template or CTL_SetType/CTL_SetTemplate makes the target types portable and named.

3.5.3 polymorph types

Portable and virtual named classes are polymorph transportable via the any type and arbitrary reference types (including pointer), see section <reference types>.

Example:
sender

```
ctl::link p;
wire::algorithmBase *alg=new wire::Algorithm<char*>;
p<<alg;
```

reader

```
ctl::connect<wire::algorithm<char*> >();
wire::algorithmBase *alg=0;
p>>alg;
```

On the readers side alg will now be a pointer to an wire::Algorithm<char*> which is a copy of the object allocated by new wire::Algorithm<char*> on the sending side.

In order to make the defined type name available to the reader the function ctl::connect<T>() must be called before reading is done.

3.6 abstract binary types

If two components have to exchange structured data types without sharing any data type declaration an abstraction concerning binary representation is needed. The mean ideas are

- any structured type is a composition of simpler types
- there are only a few compositions
- to read a structured data only its binary representation is needed

3.6.1 fundamentals

The fundamental types defined in the CTL are:
the integral types

```
bool with values in {0,1}
char=int1, int2, int4, int8
unsigned char =uchar=uint1, uint2, uint4, uint8
```

and the float types

```
real4, real8
```

where each postfix number tells the number of bytes (=sizeof) in the binary representation.

3.6.2 array

```
template<class T> class array;
```

serialisation as

```
os<<int8(vec.size());
for(int8 i=0; i<vec.size; i++)
  os<<vec[i];
```

examples

```
std::vector<T>
std::set<T>
std::list<T>
std::queue<T>
```

3.6.3 tupel

```
template<class T0, class T1=empty,..., class Tmax=empty> class tu-
pel;
T0 t0;
T1 t1;
...
Tn tn;
(n<max)
os<<t0<<t1<<...<<tn;
```

examples

```
std::complex<T> -- tupel<T,T>
std::pair<S,T> -- tupel<S,T>
struct info
{
  int n;
  float x,y;
  CTL_Type(info, tupel, (n,x,y), 3
};
  -- tupel<int,float,float>
map<key, val> = array<tupel<key, val> >
```

3.6.4 cstring

```
template<class T> class cstring;

while(!!str[i])
  os<<str[i++];
os<<T();
```

examples

```
char *
std::string
```

3.6.5 reference

```
template<class T> class reference;

if(!t)
  return os << true << int4(-1);
int4 logAddr = os.getStreamId(t);
if(logAddr>0) // t is already in the stream
  return os << true << logAddr;
```

```
os.addReference(t);
os<<false;
const char *typeName=ctl::typeName<T>(*t);
if(!typeName)
  return os<<std::string();
os << std::string(typeName)<< binary-
Size(*t) << *t;
```

examples

```
std::auto_ptr<T>
T*
```

remarks:

If in the serialisation of a argument list references to the same object occurs more than once, the object will be placed only once the stream.

If on the reading side the exact type is connected (by calling ctl::connect<T>()) the object can be read via a reference to an arbitrary base type of T.

Example sparse matrix

The following class defines two constructors accepting a sparse matrix in the ijv and in the row by row formats.

The ijv format is an array of entries of the form (i,j, value) which has the binary representation:

```
array<tupel<int4, int4, real8> >
```

The row by row format is an array of rows, where each row has an row index and a vector of entries of the form (j, val). Its representation is

```
array<tupel<int4, array<tupel<int4, real8> > > >
```

Due to restrictions of the preprocessor a type specifier in a CI may not contain a unbraced comma, therefor this types must be rewritten in the modified list syntax as

```
array<(tupel, (int4, int4, real8), 3)>
```

and

```
array<(tupel,(int4, (array, (tu-
pel,(int4, real8), 2) , 1) ), 2)>
```

This leads to the interface

```
#define CTL_Class sparse_solverCI
#include CTL_ClassBegin

#  define CTL_Constructor1 (const ar-
ray<(tupel, (int4, int4, real8), 3)> /*ijv-
matrix*/), 1
```

```
#  define CTL_Constructor2 (const ar-
ray<(tupel,(int4, (array, (tu-
pel,(int4, real8), 2) , 1) ), 2)>) /*row-by-row-
matrix*/), 1
#  define CTL_Method1 bool, solve, (ar-
ray<real8> /*x*/) const, 1

#include CTL_ClassEnd
```

Remark: If the interface will only be used by C++ one can just use typedefs like

```
typedef tupel<int4,int4, real8> ijv_item;
typedef tupel<int4,real8> jv_item;
typedef tupel<int4,array<jv_item> > row;
```

Caller and Callee can use their own representations for example

```
typedef std::pair<int,double> jv_item;
typedef std::pair<int,std::vector<jv_item> > row;
typedef std::vector<row> roy_by_row_matrix;

row_by_row_matrix A;
std::vector<double> x;

sparse_solverCI S(A);
S.solve(x);
```

3.7 cyclic data structures

If the edges of a graph data structure are written/read via references this structure may also be cyclic. Cycles are resolved by the serialisation mechanism of the CTL, see also <reference>.

Example:

Implementation

```
#include <ci/graph.ci>
#ifndef _GRAPHNODE_H_
#define _GRAPHNODE_H_
class graphNode
{
public:
  typedef graphNode* pointerT;
private:
  typedef std::list<pointerT> neighborT;
  neighborT   neighborM;
```

```
  virtual void printPriv(std::ostream&) const {}
// used to resolve cycles
  mutable bool visitedM;
  void reset() const
  {
    if(visitedM)
    {
      visitedM = false;
      for(neighborT::const_iterator nb=neighborM.begin();
        nb != neighborM.end(); nb++)
          if(!!*nb)
           (*nb)->reset();
    }
  }
public:
  CTL_Type(graphNode, tupel, (neighborM), 1);
  graphNode(): visitedM(false) {}
  virtual ~graphNode() {}
  void addNeighbor(const pointerT &node)
  { neighborM.push_back(node);}
  void print(std::ostream &os, bool first = true) const
  {
    if(!visitedM)
    {
      visitedM = true;
      printPriv(os);
      for(neighborT::const_iterator nb=neighborM.begin();
         nb != neighborM.end(); nb++)
        if(!!*nb)
          (*nb)->print(os, false);
      if(first) // reset to non-visited
        reset();
    }
  }
};
class nodeTypeA: public graphNode
{
  double valueM ;
  void printPriv(std::ostream& os) const
  { os << "nodeTy-
peA { value = " <<valueM<< "}\n"; }
public:
  nodeTypeA(double value =0): valueM(value) {}
  CTL_Type(nodeTypeA, tupel, ((graphN-
ode&)(*this), valueM), 2);
};
```

```
class nodeTypeB: public graphNode
{
  int iM, jM;
  void printPriv(std::ostream& os) const
  { os << "node-
TypeB { i = " <<iM<< ", j = "<<jM<<"}\n"; }
public:
  nodeTypeB(int i =0, int j =0): iM(i), jM(j) {}
  CTL_Type(nodeTypeB, tupel, ((graphN-
ode&)(*this), iM, jM), 3);
};
#endif
```

CI

```
#ifndef _GRAPH_CI_
#define _GRAPH_CI_
#include <ctl.h>
class graphNode;
#define CTL_Library graph
# include CTL_LibBegin
# define CTL_Function1 bool, set-
Graph, (const reference<graphNode>), 1
# include CTL_LibEnd
#endif
```

Connect

```
#define CTL_Connect
#include <ci/graph.ci>
#include <graphnode.h>
bool setGraph(graphNode* node)
{  node->print(std::cout); return true; }
void CTL_connect()
{
  ctl::connect<nodeTypeA>();
  ctl::connect<nodeTypeB>();
  graph::connectID<1>(setGraph);
}
```

Client

```
void callGraph()
{
  graphNode::pointerT A,B,C,D;
  A = new nodeTypeA(3.14);
```

```
  B = new nodeTypeA(2.71);
  C = new nodeTypeB(3,4);
  D = new nodeTypeB(7,1);
// build a graph containing cycles (A->A), (A->B-
>C->D->A)
  A->addNeighbor(A);
  A->addNeighbor(B);
  A->addNeighbor(D);
  B->addNeighbor(C);
  B->addNeighbor(C);
  B->addNeighbor(D);
  C->addNeighbor(B);
  C->addNeighbor(D);
  C->addNeighbor(0);
  D->addNeighbor(A);
  D->addNeighbor(C);

  A->print(std::cout);
  ctl::connect<nodeTypeA>();
  ctl::connect<nodeTypeB>();
  graph::use("../../graph/linux/graph.exe -
l tcp");
  graph::setGraph(A);
}
```

4 CI Classes

CI classes are named and portable.

4.1 CI class as an extrinsic interface

The g++ Compiler supports te so-called signature concept. This concept is also represented in the CTL, If a class Impl implements all constructors, method and static methods given in an Interface CI then

```
Impl impl;
CI ci(impl);
```

instantiates a CI class which implementation is given by the Impl impl. The object ci will hold a copy of ci, which will be got by the Impl copy constructor.

5 Creation of a service

makefile using g++ on a linux system

```
CC    = g++
CTL   = $(CTL_PATH)/ctl/include
CI    = $(CI_PATH)
LIBS  = -limpl.a -ldl -lpthread #-llam -lpvm
CFLGS = -I($CTL) -I$(CI) #-DCTL_MPI -DCTL_PVM -
DCTL_LOCATE= $(MY_SELECT)

service.exe: connect.o makefile
  $(CC) -shared connect.o $(CLIBS) -o service.so
  $(CC) connect.o $(CLIBS) -o service.exe

connect.o: connect.cpp makefile
  $(CC) -c $(CFLGS) connect.cpp -o connect.o
```

6 Types of Linkage

The component technology implemented by the CTL can be seen as an generalisation of dynamic linkage to a runtime linkage with different link mechanisms.

For each linkage type the same following syntax is valid

```
ctl:location loc(”user@host:path/exec [link-
age][args]”);
ctl::link P(loc, [linkage]);
```

As the service for the cases lib and thread a shared object service.so is needed. The other linkage types need an executable like service.exe.

6.1 Supported Linkage Types

6.1.1 lib

Behaviour like classical static linkage. In the time frame of the caller the result is immedatly available.

6.1.2 thread

Start in a seperate thread each function or method invocation. The link creates one thread for execution. Subsequently called functions will be queued.

6.1.3 tcp

use sockets for communication

The connections are buildt up in the following steps

- the client spawns service by [ssh] service.exe tcp:host:port:logID:nprocs and invokes accept(port)
- the service invokes connect(host) to establish the connection

6.1.4 pipe

the client invokes service by

[ssh] service > inpipe < outpipe

where inpipe and outpipe are cretaed by the client.

Note: this mechanism works also through a firewall.

6.1.5 pvm

assumes te compile flag CTL_PVM

uses pvm for communication

6.1.6 mpi

assumes te compile flag CTL_MPI

uses mpi for communication

6.1.7 dynamic mpi groups using lammpi

like mpi but assumes the lam mpi-version for dynamic process allocation

6.1.8 daemons

uses sockets for communication

start a service optionally telling him a port to listen

```
service.exe [port]
```

If no port is given the service will use an random port number. Now the service will wait for connection using the given or random port number.

Any process can create a link to this service by

```
link daemon(”host:port dmn”);
```

If not denied by a firewall the service will accpet a connection to the client.

This connection exists as long as the link at the clients side.

6.1.9 file

link str(filename mode , file)

Remark the cases thread, tcp, pvm, mpi, lam, daemons have all the same behaviour concerning result availability.

6.1.10 summary of linkage types

The following table gives an overview of the supported linkage types.

channel	used tools	communication via:	creation via:.	connection via:
lib	libdl	function call	dlopen	dlsym
thread	libpthread	function call	dlopen+pthread_create	dlsym
tcp	sockets+ssh	tcp/ip	[ssh] spawn	accept:connect
pipe	pipes+ssh	stdin/stdout	[ssh]spawn	pipes
mpi	mpich/lammpi	MPI_Ibsend/MPI_Recv	[ssh] mpirun	MPI_Init
lam	lammpi	MPI_Ibsend/MPI_Recv	MPI_Comm_spawn	MPI_Init
pvm	pvm	pvm_send/pvm_recv	pvm_spawn	pvm
dmn	sockets	tcp/ip	—	connect:accept
file	stdio	fwrite/fread	—	fopen

channel	location	local	remote	group
lib	path/lib	yes	no	no
thread	path/lib	yes	no	nyi
tcp	user@host:path/exe	yes	yes	dynamic
pipe	user@host:path/exe	yes	yes	no
mpi	host:path/exe	yes	yes	static + extrinsic
lam	host:path/exe	yes	yes	dynamic
pvm	host:path/exe	yes	yes	dynamic
dmn	host:port	yes	yes	no
file	filename	yes	no	no

The linkage *lib* corresponds to dynamic binding; an addressed function of the component is evaluated synchronously inside the calling process.

Also the linkage *thread* leads to a local execution of the addressed function, but asynchronously in a separate thread.

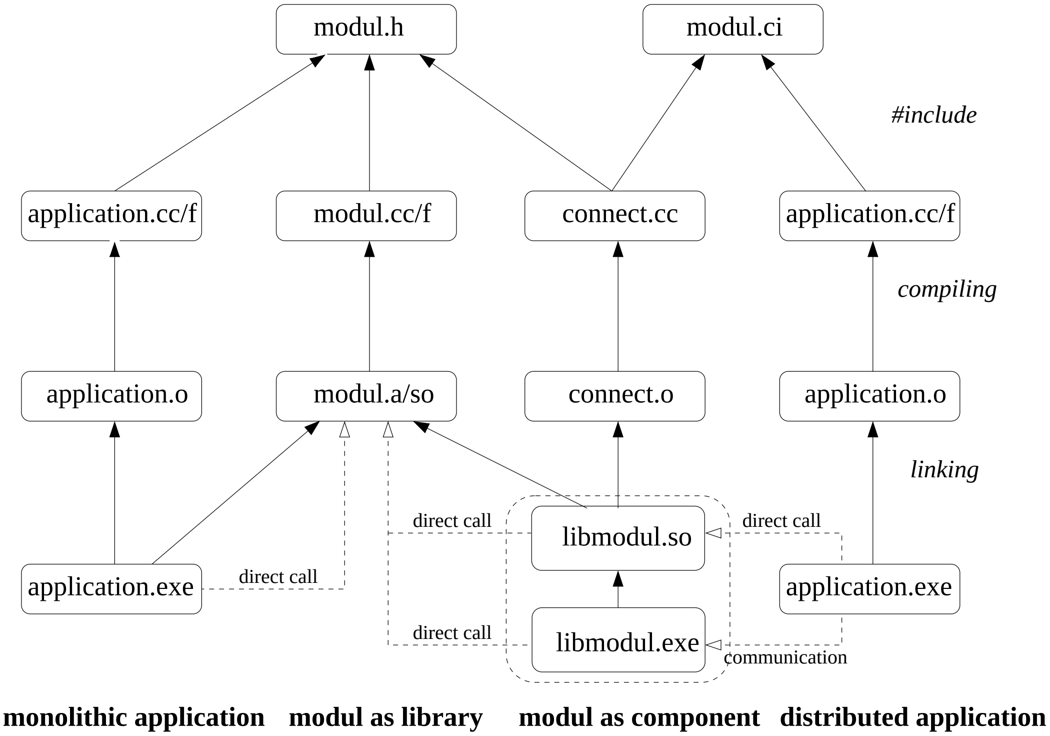
The other specified linkage variants permit asynchronous remote execution of functions. Up to the *dmn* linkage, with which a TCP/IP connection to an existing process is build, here new processes are started on. Using the *pipe* linkage it is possible to connect through a Fire Wall to the remote processor. Using the *mpi* linkage an MPI application scheme (in itself parallel) can be used as a CTL component. The different linkage variants are thereby in the sense transparency that the kind of the use is independent in syntax and program flow of the chosen linkage.

6.2 Comparison static linkage versus CTL linkage

One main concept of the CTL is a generalisation of linkage.

In the monolithic case an application is build up by the linker from a list of objects and dynamic or static libraries. For each called function the compiler wants to see its declaration. After compiling the linker first looks in the given list of objects and libraries for an definition of the called function (using it's signature as an identifier) and then binds the call to exactly one implementation. In a list of objects such a definition may occur only once (otherwise multiple definition), in a list of libraries the first found imlementation will be taken. If no definition was found, the error ”undefined symbol” occures.

In this case all listed objects and libraries must be available at linkage time on the compiling machine. Furthermore the set of used implementations is determined already at linkage time. While run time this implementation will be executed on the same processor in the same process as the calling function.



The Windows registry and the Unix/Linux dlopen mechanisms enable run time linkage. But also here only one implementation of a function can be used, the execution is still performed on the same processor and in the same process.

The CTL gives both, the selection of an implementation and the binding, in users hand. At run time it can (and must) be selected which implementation on which host to be linked in which mode. In the definition of the library (in this context called component) the binding of the function signature to an implementation must be given.

The above figure shows the dependence graph during classical connection (left) and component connection (right). While in the first case the application depends directly on the used module, it depends in the second case only on the interface (modul.ci). Consequences of it are:

* The application does not have again to be recreated, if the module is modified.

* At run-time the application can decide, which components and how many instances of these components on which hardware to be used.

6.3 process groups

The class ctl::group implements the Single program multiple data (SPMD) and the Single program multiple data MPMD programming models.

6.3.1 client

container<location> locations container in { std::vector, std::list, std::set, ctl::vector }

```
group(locations, ctl::linkage)
```

constructor creates complete graph of the processes defined by the locations.
Location from { pvm,lam, tcp }
The creating process get's in the group the logId 0.

```
std::list<ctl::location> list;
list.push_back("../../group/linux/group"); list.push_back("../../group/linux
ctl::group G(list, ctl::tcp);
int num = 100;
int sum = G.globalSum(num);
```

6.3.2 service

```
group(int argc, const char *const * argv, const char *log-
File, bool cwd)
```

example

```
#include <ctl.h>
int main(int argc, char ** argv)
{
  ctl::group G(argc, argv);
  int me = G.logId(), nProcs = G.size();
  int sum = G.globalSum(me);
  printf("me = %d sum = %d\n",me, sum );
  if(nProcs > 2 && me==1)
  {
    double x = 3.1416;
    G[2]<<x;
  }
  if(nProcs > 2 && me==2)
  {
    double x;
    G[1]>>x;
    printf("got x = %e\n",x);
  }
  return 0;
}
```

call by mpirun -np 4 group mpi or by client

or by other code via

6.3.3 global operators

global assoziative operations are performed in the hypercube topology, see ???

{add, mult, min, max, sample, {udf}}

6.3.4 termination of asynchron recursions

group::run starts a termination algorithm, see Mattern.

7 Life time of CI-object and processes

7.1 life time of processes

The life time of processes is defined via ownership.

The creating process is the owner. The child process will terminate, when the referencing group or link is destroyed. This is the case at the latest when the owner terminates.

7.2 life time of objects

The life time of objects is defined via reference counting using the credit method, see Mattern. If the last reference is destroyed the object will be destroyed. This need one remote method invocation at the time the last reference inside one process to an CI-Class is destroyed.

8 binding of CI's to processes

In the order of priority the following four mechanisms for binding a function, constructor or static method call to a link exist.

8.1 selection by first argument

To any global function, constructor or static method of a CI, the destination link can be optionally given as the first argument

```
int main()
{
  // as before
  firstClass C(firstProc, “initfile”);
  std::cout<<C.f(x)<<std::endl;
  std::cout<<firstCI::f(firstProc, x)<<std::endl;
```

```
}
```

Remarks:

A link is not portable and may not be an argument in the CI definition. Therefore overloading ambiguity will not arize.

Non static methods do not accept a link as the first argument. The destination link is here determined by the corresponding CI class object.

8.2 temporary selection

```
int main()
{
  ctl::location  loc(“host:/usr/peter/comp/firstImpl.exe tcp”);
  ctl::link p(loc);
  firstCI::use(p);
  firstClass C(“initfile”);
  firstCI::use();
}
```

see also <global functions> : use

8.3 local selector

Define a locator by

```
ctl::location my_locator(const std::string &id, const ctl::any &prop-
erty);
```

or

```
class my_locatorType
{
  ...
  ctl::location opera-
tor() (const std::string &id,   const ctl::any &prop-
erty);
} my_selector;
```

and give it to the ctl environment by

```
int main()
{
  ctl::setLocator(my_locator);

  firstClass C(“initfile”);
  std::cout<<C.f(x)<<std::endl;
}
```

8.4 global selector

bash > export MY_LOCATOR=”/usr/peter/comp/my_selector.so lib”

Compile with the define like

g++ -DCTL_LOCATE=MY_LOCATOR appl.cc

```
int main()
{
// just use firstClass
  firstClass C(“initfile”);
  std::cout<<C.f(x)<<std::endl;
}
```

9 Usage by other languages

9.1 Usage of the C/Fortran Interface

The CTL provides language bindings for C and FORTRAN, eg. it is possible to transform an existing FORTRAN library into a remote accessible software component and to use a CTL component by Fortran applications. Especially into Fortran only a subset of the C++ features can be mapped. Therefore only a restricted set of CI’s can be used or implemented by Fortran. The restrictions are

- only CI classes, no CI libraries,
- no constructors,
- no operator nor method overloading,
- only methods returning void and accepting argument types listed in the following section.

9.1.1 Parameter Conversion

In the standard Fortran programing language no structured types like string or array (= vector) of a type are expressible.

On the other hand such structures are needed to implement a type safe message passing.

Therefore the interface is defined in such structures like an array which aggregates the information about number, type of data and the data itself to be transmitted between different processes. Hence, a translation of these structures to fundamental Fortran data types is needed.

Translation to Fortran Types:

interface type	⟹	**fortran representation**
	fundamentals	
char	⟹	**character**
int2	⟹	**integer*2**
int4	⟹	**integer*4**
int8	⟹	**integer*8**
real4	⟹	**real*4**
real8	⟹	**real*8**
	structures	
string	⟹	**integer*8, character**
array<interface fund.>	⟹	**integer*8, fortran fund.**

For example the interface signature
void firstClass::f (const real8, array<real8> &)
has to be implemented by
subroutine firstClass_f_impl(accuracy, sx,x)
real*8 accuracy
integer*8 sx
real*8 x(sx)

9.1.2 Functions for construction, destruction and result handling.

create a new CI object C: void new_CI(char *rlib, char *path, char *host, int *simu_handle)
Fortran: subroutine new_CI(rlib,path,host,simu_handle)
character* rlib
character* path
character* host
integer*4 simu_handle

if the zero terminated strings rlib, path and host describe a valid name and location of a simulation service, new_simu will start the simulation path/rlib on the machine host and simu_handle becomes a positive handle for this simulation process.

On failure simu_handle is set to 0.

If host is the empty string '\0', the localhost is chosen.

If rlib is the empty string a local simulation object is created in the current process. In this case the corresponding service object must be linked to the current executable.

release a ci object C: void rel_CI(int ci_hande)
Fortran: subroutine rel_ciname(ci_hande)
integer*4 ci_hande

If ci_hande is a valid handle (obtained by a call to new_ciname) the dedicated ciname-process will terminate.

Further calls of ciname_<method>(ci_hande, <param>, res_handle) – see below – will have no effect apart from that ci_hande and res_handle will be set to 0.

All result handles obtained via ci_hande become invalid.

receive the result(s) of a former ciname call C: void recv_ciname(int *res_handle)

Fortran: subroutine recv_ciname(res_handle)

integer*4 res_handle

If neither res_handle nor -res_handle is a valid handle obtained by a call of the kind ciname_<method>(ci_hande, <param>, res_handle), res_handle is set to -2.

If res_handle was given as a negative value of a valid handle, recv_ciname waits for the result answer, then sets res_handle to -1 and finally writes the result values into the variable(s) given before (via <param>) to the corresponding call of ciname_<method>(ci_hande, <param>, res_handle).

If res_handle is positive (and a valid result handle) recv_ciname tries to receive the result answer. If this is already available the result is written into the variable(s) as above and sets res_handle to -1. If the result is not yet available res_handle is unchanged.

IMPORTANT:

Please ensure that at the time recv_ciname is called, the variables in <param> are still in scope.

Otherwise your stack will be corrupted!!

call a ciname method C: void ciname_<method>(int *ci_hande, <param>, int *res_handle)

Fortran: subroutine ciname_<method>(ci_hande, <param>, res_handle)

integer*4 ci_hande

<param> Fortran parameter of <method>

integer*4 res_handle

If ci_hande is not a valid handle obtained by a call to new_ciname or if the ciname-process died in the mean time, ci_hande will be set to 0 and the call has no further effect.

Otherwise the method <method> of the cinamelation dedicated by ci_hande is called with the parameter <param> and ci_hande is unchanged.

If either the result is already available (due to a local cinamelation object, see under new_ciname) or no result is expected (all parameter in <param> are const) res_handle will be set to 0.

If res_handle was given as a negative number, this call waits for the result answer and sets res_handle to -1.

In all other cases res_handle is set to a positive number which can be used to receive the result later on using recv_ciname.

9.2 Java

There is already a prototyp of a Java environment using reflections which embedding Java classes into the CTL framework via tcp/ip.

9.3 preprocessor

In the current CTL implementation the C-preprocessor is used to generate from the CI definition the functions needed to perform the remote invocations. This has the advantage that no other tool than the C++ -Compiler is needed to generate CTL components.

Obvious disadvantages are the limitations of the preprocessor and the strange compiler messages if inside a macro expansion a compile time error occurs.

A consequence of the first point is the CI syntax the number of arguments in a list must be explicitly given and that a type specifier may not contain a comma, see the <CI-grammatic>. A seperate CI syntax checker circumvents the second point.

10 Grammar of the Component Interface

The CTL interface description is defined in terms of C-macro definitions which are expanded to the functions which perform the data serialisation and transport. Due to the fact, that hte C-preprocessor can not count lists, at end of each declaration the number of arguments must be given. The following grammatic supports the C++ features of ...

ident → valid C identifier
numargs → integer in {0, ..., maximal Args}counting the entries in a list
id → integer in {1, ..., maximal ID }representing a function ID
op → overloadable C operator
funcname → ident | **operato**r op
type → [**const**] ident[*] | [**const**] ident<type> **|** **(**[**const**] ident**,(**type [**,**type] **),** numargs**)**
typelist → **()** [**const**]**, 0** | **(** type [**,**type] **)** [**const**]**,** numargs
funcsign → type **,** funcname **,** typelist
constructor → **#define CTL_Constructor**id typelist
[**#define CTL_Constructor**id**Throws** typelist]
method → **#define CTL_Method**id funcsign
[**#define CTL_Method**id**Throws** typelist]
staticmethod → **#define CTL_StaticMethod**id funcsign
[**#define CTL_StaticMethod**id**Throws** typelist]
function → **#define CTL_Function**id funcsign
[**#define CTL_Function**id**Throws** typelist]
functiontmpl → **#define CTL_FunctionTmpl**id funcsign **,** typelist
[**#define CTL_FunctionTmpl**id**Throws** typelist]
classentry → method | staticmethod | constructor
classbody → **#include CTL_ClassBegin**
classentry [classentry]
#include CTL_ClassEnd
class → **#define CTL_Class** ident
classbody
classtmpl → **#define CTL_ClassTmpl** ident **,** typelist
classbody
libraryentry → class | classtmpl | function | functiontmpl
librarybody → **#include CTL_LibBegin**
libraryentry [libraryentry]
#include CTL_LibEnd
library → **#define CTL_Library** identifier
librarybody
interface → library | class | classtmpl