# EUROPA Parallel C++
# Version 2.1

The EUROPA Working Group on Parallel C++
Architecture SIG

July 10, 1997

Edited by
Caromel Denis (caromel@unice.fr)
McEwan Alistair (A.McEwan@lpac.ac.uk)
Nolte Joerg (jon@first.gmd.de),
Poole Jonathan (jpoole@sophia.inria.fr),
Roudier Yves (roudier@unice.fr),
Sagnol David (dsagnol@sophia.inria.fr),
and the Europa group
Challier Jean-Marc (challier@irit.fr),
Dzwig Peter (P.E.Dzwig@lpac.ac.uk),
Kaufman Richard (richard@ilo.dec.com),
Liddell Heather (heather@dcs.qmw.ac.uk),
Mussi Philippe (Philippe.Mussi@sophia.inria.fr),
Parkinson Dennis (dennisp@dcs.qmw.ac.uk),
Rigg Malcolm (malcolm@sst.icl.co.uk),
Roberts Graham (G.Roberts@cs.ucl.ac.uk),
Winder Russel (R.Winder@dcs.kcl.ac.uk).

**Abstract**

This paper presents the definition of EUROPA: a framework within which parallel C++ environments can be developed and standardised.

EUROPA (also called EC++) sets out a framework which will add portability to parallel C++ systems and will run across a variety of hardware architectures, while encompassing as wide a set of parallel computing models and paradigms as possible, both standard models and user extensible models. This is done entirely within standard C++, i.e. without syntactic extensions to C++.

# Contents

# List of Figures

# 1 Introduction

## 1.1 Structure of the Working Group

The EUROPA Working Group on parallel C++ has been formed with a view to defining a standard framework for parallel C++ systems. The members of this Working Group come from a variety of backgrounds—researchers into parallel C++, hardware vendors, compiler writers and end-user application developers. Input has also been taken from other (non-participating) parallel C++ users and researchers.

## 1.2 Aims of EUROPA

### 1.2.1 Reconciliation of Approach

The aim of the standardisation work is to reconcile the various approaches to parallel C++, and to define a framework within which parallel C++ languages can be expressed and standardised.

### 1.2.2 Portability

This will make parallel C++ systems more portable both across hardware architectures and across compiler implementations, thus giving application developers the portability of code that is needed. It is not the aim of this work to attempt to standardise the number of computing models and paradigms that exist within the parallel C++ community.

### 1.2.3 Implementation of New Models

Rather, EUROPA (also called EC++) will provide a standard framework where existing models can be implemented in a portable and efficient manner, and new-, user-, application-specific paradigms and models can be designed in a standard context.

### 1.2.4 Language Definition

EUROPA will not define a parallel C++ language per se, or indeed an exhaustive set of parallel semantics per se; rather it will seek to define a set of standards and guidelines which language designers and compiler writers will need to consider and follow in order to have a parallel C++ language which can be said to conform to the standard. It is this framework for building these systems which is seen as the standard, rather than defining specific content of these systems thenselves. In language syntax terms, the standard is as defined by ANSI/ISO C++ and EUROPA does not change this.

# 2 Rational of EUROPA

## 2.1 Syntax and Semantics

EUROPA assumes the syntax and semantics of C++, and allows a set of semantic extensions to allow parallel code to be constructed and to allow parallelism to be expressed. There is no allowance within the EUROPA definition for any syntactic extensions to C++. This means that the syntax of EUROPA is precisely that of standard C++, and that sequential C++ code is a subset of EUROPA, therefore allowing standard sequential C++ code to be included in the EUROPA definition.

## 2.2 Hardware Architectures

EUROPA will provide a framework for parallel C++ systems across a variety of hardware architectures. The only assumption about hardware architectures is the minimalistic view that they will consist of a set of basic computing resources (processors) with some form of associated memory and storage.

The resources themselves can be massively parallel systems, multi-processor machines, distributed systems single processor systems or any other architecture (as a result of Section 2.1).

No assumption is made within the EUROPA model that these resources are unique processor/memory pairs, shared memory processors or otherwise.

EUROPA will offer the user the possibility of a view of a single system which will hide the details of the architecture of the (potentially complex) systems underneath.

## 2.3 Control of Parallelism

### 2.3.1 Usage Patterns

Differing applications, problem domains and parallel processing models require different control abilities over the parallel constructs used in a program. These control requirements largely define the usage patterns of the language in question and will vary with the application in question. EUROPA will support variants of common classes of C++ applications and with them any assumptions about usage patterns.

### 2.3.2 Control Assumptions

EUROPA is a generic standard for different parallel C++ languages and implementations, no assumptions can be made about any one specific control mechanism. Instead, a means of allowing different control mechanisms within different parallel C++ languages should be possible, constrained only by any restrictions which C++ would impose on those control mechanisms.

## 2.4 Definition of Programming Models

The definition of programming models is covered in Section 5, the proxy layer of the standard. This section gives examples of standard programming models and for user extensible models.

No assumption is made within the EUROPA definition about any communications implementations, due to the range of hardware architectures and the range of programming models that EUROPA is attempting to cover.

### 2.4.1 Communications Assumptions

Objects within a parallel C++ system will require to communicate with each other to allow useful parallel code to be written, either in method invocations, in value passing. As no assumption can be made about the actual transport layer, standard and generic communication interfaces such as PVM and MPI can be used. This communications interface need not be seen by application writers.

The designers of parallel programming libraries can choose the communications interface most appropriate with their model, thus still maintaining a level of compatibility and portability.

## 2.5 Systems libraries

### 2.5.1 Systems Library Facilities for Parallel Processing

The EUROPA definition will contain a further library definition which will define and standardise the systems library facilities required for parallel processing models. These libraries will define sets of library calls which applications programmers can reasonably expect to be supplied with a particular implementation of the EUROPA system.

### 2.5.2 Behaviour of Systems Libraries

Some of these library functions may be common to all platforms, while some could be specific to individual platforms. The behaviour of the libraries will exhibit the same behaviour in any model independent of the architecture upon which the model is being implemented. Systems libraries will include the standard ANSI/ISO libraries.

### 2.5.3 Systems Libraries Tools

Systems libraries will provide tools which encapsulate details and knowledge of the architecture of a resource; and which can make this information available to the proxy system.

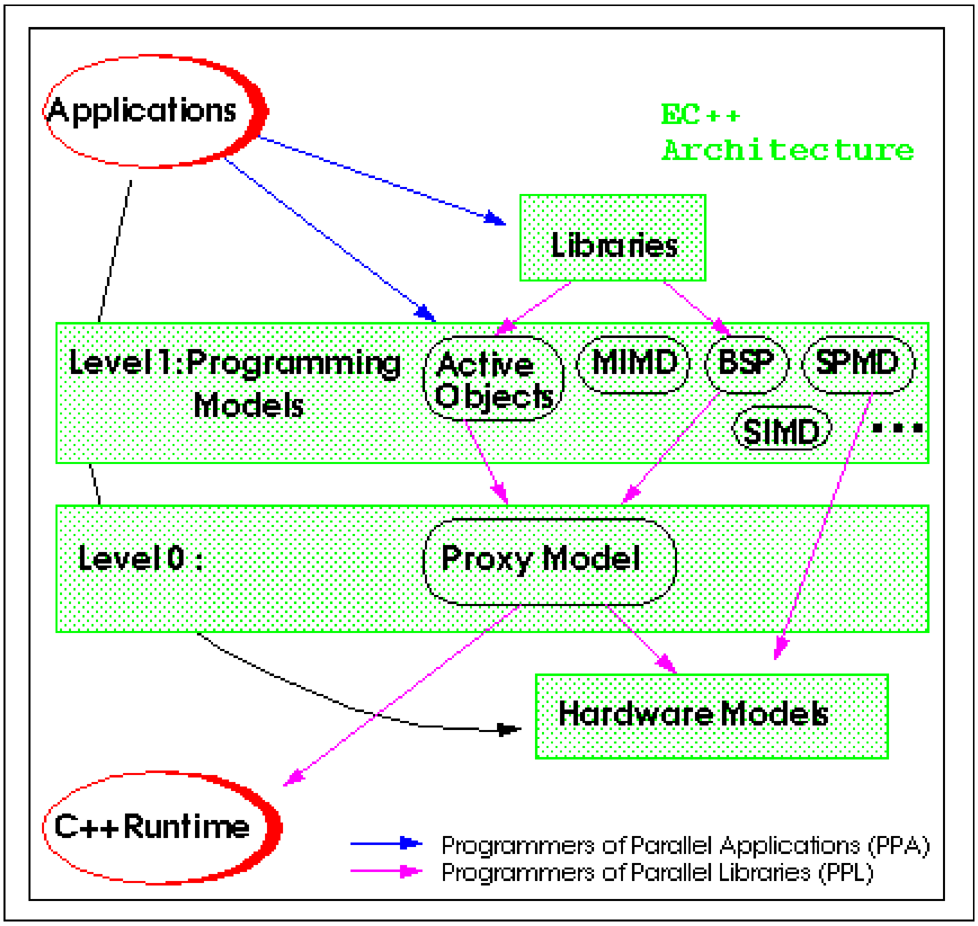


Figure 1: Structure of EUROPA standard

# 3 EUROPA Model and Structure

The EUROPA model proposes the following structure (Figure 1):

- A Proxy Model (Level 0) [Section 4]
- A Set of Programming Models (Level 1); [Section 5]
  - Standard models e.g. MIMD, SPMD, BSP, etc.;
  - User extensible models.
- A Standard Set of Libraries;
  - PVM, MPI libraries/bindings;

  - Math libraries etc.;
  - Parallel I/O;
  - Tool libraries (e.g. Performance and Correctness, Debugging tools, Load balancing, etc).

- A Set of Hardware Models/Systems libraries (Level 2);
  - Interface to hardware resources: shared memory etc.;

Level 0 is dedicated to *programmers of parallel libraries*, while Level 1 frameworks being build with the standard are targeted towards *programmers of parallel applications* (Figure 1).

## 3.1 The EUROPA Runtime Model

Figure 3 shows in a diagrammatic form the basic behavior of the runtime model.

With the proxy model being independent of any parallel paradigm, it permits the building of libraries of concurrent programming models. The hardware model describes various machine specific primitives, and permits portable implementations of user extensible models.

## 3.2 The Applications View

End-users, i.e. programmers of parallel applications (Figure 1), are concerned with application parallelism, and as such see and use the various high level *Programming Models*, and specialised *Libraries*. In contrast to this, programmers of parallel models and paradigms, i.e. implementors of EUROPA are concerned with the *Proxy Model* and the *Hardware Model*.

## 3.3 Development of Programming Models

The EUROPA framework makes it possible for applications programmers to develop various application models, some very general, some very specific and highly targeted, thus allowing EUROPA to address a wide range of application domains.

## 3.4 The EUROPA Compilation Model

The compilation scheme of EUROPA is presented in Figure 2: it emphasises the important point that no changes on the system compilers or tools are involved.

### 3.4.1 Pre-Processing requirements

Only non-intrusive pre-processing (the proxy generator) is required. It is important that this pre-processing is non intrusive, that is to say that it does not change applications code, but rather only generates extra C++ code to be compiled with standard C++ compilers.

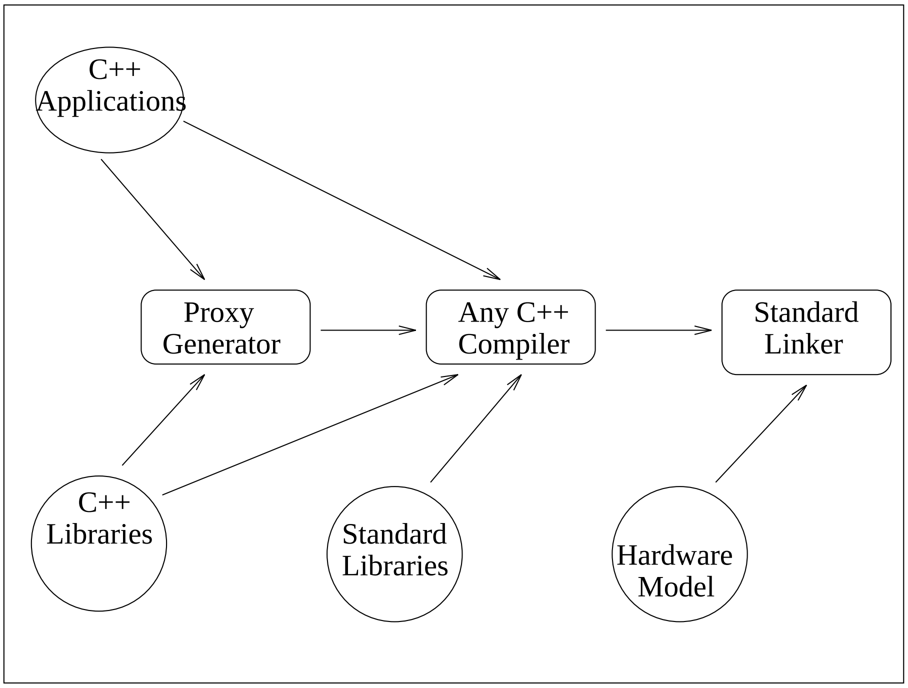


Figure 2: EUROPA compilation scheme: standard compilers and tools

# 4 EUROPA Level 0: The Proxy Model

Reader is invited to refer to Figure 1 for the overall architecture of the EUROPA standard.

The proxy model (Level 0) permits the specification of *proxy objects* for standard C++ objects; it is based on meta-object techniques. Associated tools automatically generate the C++ code of the proxies. Based on this capability, libraries for parallel programming can be build in a uniform manner (Level 1). In the following sections we detailed the proxy model.

## 4.1 Proxy generation

### 4.1.1 The EC_Reflect Class

The *EC_Reflect* class allows to define classes that will be instantiated with a meta-object (a proxy); such an object is called a *reified object* (Figure 3).

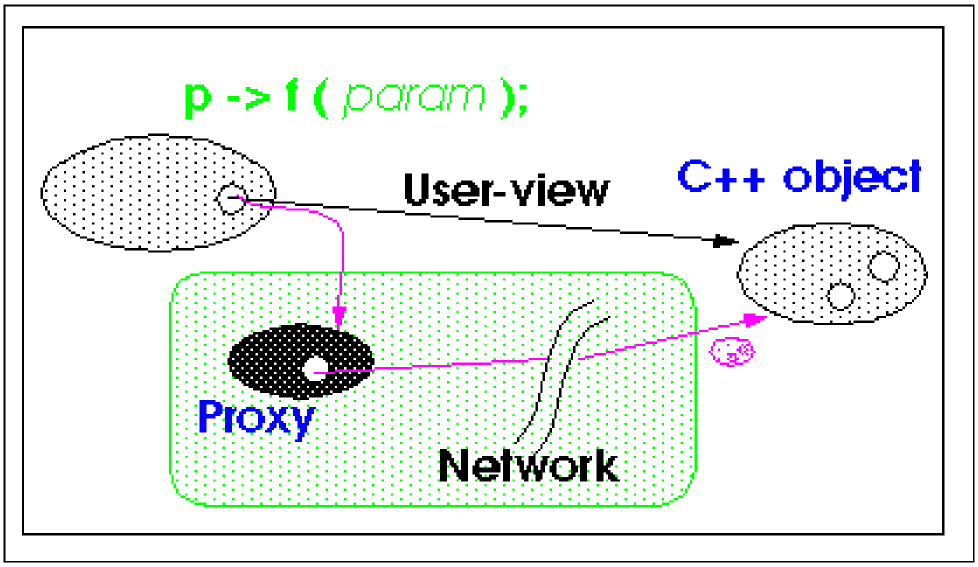


Figure 3: Principle of reification

*EC_Reflect* derived classes allow the definition of both object and proxy together, within the same class structure. It permits the end-users to use the proxies in a transparent way, just by inheriting from library classes in the usual way.

All the public calls issued to a reified object through a pointer, will trigger the execution of the method *ec_reify* of the proxy with the appropriate object of type *EC_Call* as a parameter.

```
namespace EC {
```

```
    class EC_Reflect {
    public:
      void* operator new(size_t s, etc.);
         // Optional

    protected:
      virtual EC_Proxy* ec_proxy();
         // Definition of the proxy object:
         // class being instantiated as the meta-object (proxy).
         // Return type to be overridden by derived classes with
         // specific proxy class
    };
}
```

However, public member functions that statically show an access to the member function *ec_proxy()* are not reified, but directly called on the proxy.

### 4.1.2 The EC_Proxy Class

The *EC_Proxy* class is the only part of a reified object being instantiated automatically when such an object is created; the proxy object is instantiated locally, while the reified object can be instantiated remotely. Users are expected to define the proxy classes they need, inheriting from the EC_Proxy base class.

```
namespace EC {
    class EC_Proxy {
            // To be derived by EUROPA users as  proxy classes
    public:
        EC_Proxy (...Allocator params..., EC_Class cl, EC_Call* c) {
            // Proxy constructor
            // Called with reified object allocator parameters,
            // id of class to be created, reification of constructor
        }

        virtual void ec_reify (EC_Call* c) { // A call reification
            // Called upon calls issued to the reified object
        }
    };
}
```

For all classes that directly or indirectly inherit from *EC_Reflect*, upon a dynamic allocation of one of the two forms (EUROPA implementation dependent):

```
robj=new(...allocator params...) Derived_EC_Reflect(...constructor params...);
or
robj = ec_new ( (allocator params), Derived_EC_Reflect, (constructor params));
```

the following actions take place:

- an object of the type of *ec_proxy* is instantiated locally,

- the constructor of *ec_proxy* being called with the following parameters:
  - the allocator parameters if any,
  - an object of type *EC_Call* representing the constructor parameters above,
  - an object of type *EC_Class* representing the class *Derived_EC_Reflect*.

All the classes publicly inheriting from *EC_Reflect* present this behavior.

Please note that there must be a conformance between the allocator parameters of a reified class, and the constructor of the proxy being used for it. On a typical use of EUROPA presented Figure 4, the conformity that must hold is:

`Const. params of` *Rem_Proxy* `= Alloc. params of Remote + EC_Class + EC_Call`

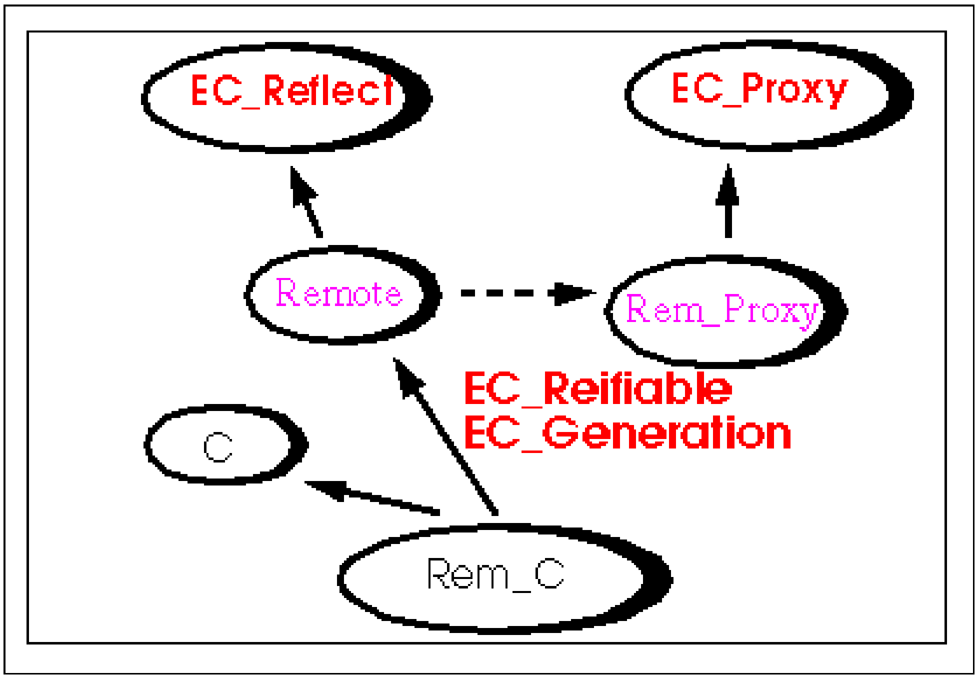


Figure 4: Typical use of EUROPA

### 4.1.3 The EC_Call Class

Upon each occurrence of a call on a reified object, e.g.

```
robj->foo(params);
```

an object of type *EC_Call* will represent the call being made, and be a parameter of member *ec_reify*, in classes derived from *EC_Reflect*.

```
namespace EC {
    class EC_Call {
    public:
        EC_Any* eff();  // Constructs a list of effective parameters
        EC_Mb_Fct mb(); // member function to be called
        EC_Any res;     // address to put the result
        void execute (EC_Any obj);
              // Execute the member with the current effective parameters,
              // on the object obj; puts the result in res member.
              // Overriding resolution (Dynamic Binding),
              // but no overloading resolution here.
        EC_Call* clone ();
              // Return a shallow copy of the object
    };
}
```

Such an object is dynamically allocated; its lifetime extends only up to the end of the *ec_reify* function. The *clone* member can be used to get a copy if needed.

## 4.2 Object and class representation

An important feature is the capability to automatically achieve (un)marshalling of objects between processes and address spaces.

### 4.2.1 The EC_Any Class

This first class allows for handling of C++ values (fundamental, objects, etc.) in a generic manner. The types associated with an object is always the dynamic type.

For a class $A$, the object can be of type $A\&$, $A$, $A*$, $A**$, etc. (implementation dependent). When the object is reified, it means that there is a proxy associated with the object itself, denoted by the function *is_reified*. The type then is, for instance, "*Reified* $A*$" which is compatible with an $A*$, and all ancestors of A; it can be seen as a derived class of $A$.

```
namespace EC {
    class EC_Any {
    public:
            EC_Any ();
            EC_Any (const EC_Any&);
            ~EC_Any ();
            EC_Any &operator= (const EC_Any&);

            EC_Class cid(); // get class id

                // Any creation
            EC_Any (*** all basic types ***);
            EC_Any (EC_Class c, void *value);
```

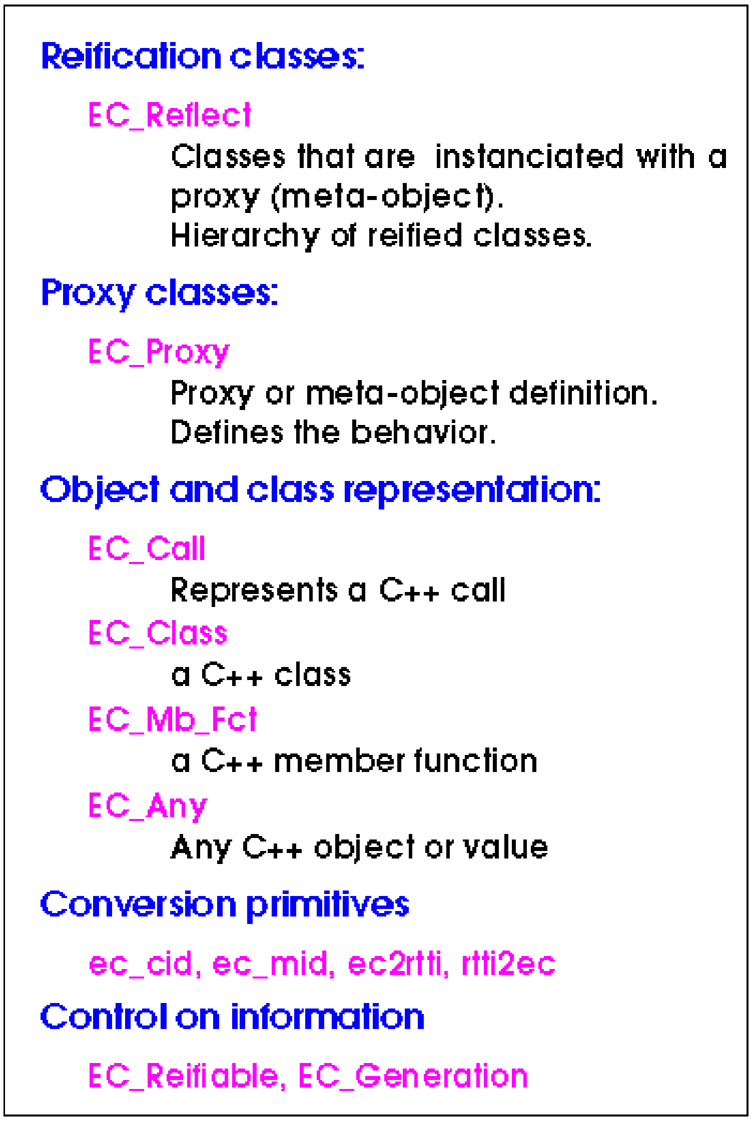


Figure 5: Basic classes of EUROPA Level 0

```
          // value must be consistent with c

    // Extraction
bool operator>>=(const EC_Any&, *** all basic types ***);
bool operator>>=(const EC_Any&, *** all EC_Class types ***);

bool is_null(); // Check if a pointer is null
```

```
                // Advanced functions for dynamic control of reification
            EC_Proxy* is_reified(); // Returns the proxy if the current
                                    // object is reified. Null otherwise.
            void reset_reified(EC_Proxy* p);
                      // Change the value (proxy, and everything below)
                      // of a reified object. Keep the same type.

            // Modification of pointer or reference types of current object,
            // returns false if operation is illegal
      bool deref ();    // dereference a pointer object
      bool derefs ();   //       upto the object without pointer
      bool addptr ();   // add a pointer to the object
   };
}
```

The EC_Any class is used as follows:

```
int i=14;
A* a=new A ();
EC_Any any1(i), any2(a);
...
int j;
A* a2;
if ( any1 >>= j) {
     // use the value j
};
if ( any2 >>= a2 ) {
     // use the value b
};
```

#### 4.2.2 Class representation: EC_Class

All C++ types (fundamentals and compounds) are uniformly represented through the *EC_Class* (int, float, etc., array, struct, and, of course, classes). This class is both a *meta-class* for the representation and manipulation of C++ objects, and (by construction) a *type code* that can be used as an id for types.

Function members such as *mb_nb(), mb_name(), mb_get(), etc.* provide information on the class structure, and access to the member data. Thanks to these features, generic and optimized algorithms for (un)marshalling of objects can be defined.

For fundamental types, *mb_nb()* is equal to 1, and functions *mb_nb(), mb_name(), mb_get(), etc.* are valid. The enum *EC_Type* below provides for *switch* discrimination of fundamental types.

For C++ arrays, in the case of a static array, *mb_nb()* gives the number of element, and *mb_get()* provides access to them. For unknown size arrays, *mb_nb()* returns 0.

Finally, a set of functions allows the manipulation of pointer or reference types.
*addref()*, applied on a type $A$ returns the *EC_ Class* for *&A*;
*ptr()* applied on $A$ again returns $*A$, while applied on *&A* it returns $A$;
*deref()* returns $A$ when applied on $*A$,
while *derefs()* remove all the pointer access upto a non-pointer type (returns $A$ on $*...*A$).

All these functions return 0 if operation is illegal, or reaches the implementation limits (e.g. no more than $***A$ is represented).

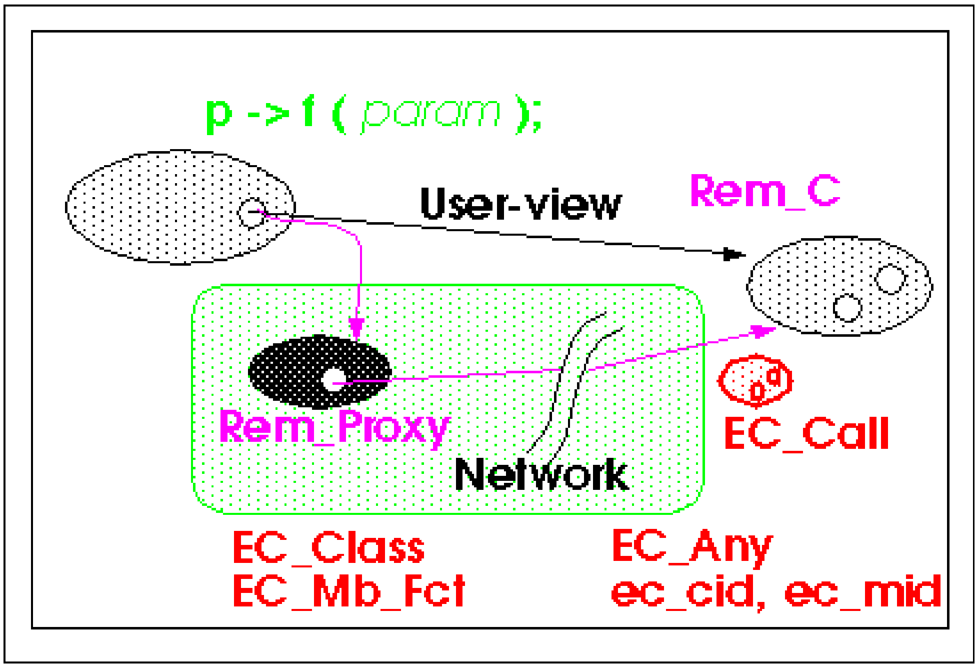


Figure 6: Runtime view of reification with basic classes

```
namespace EC {
    class EC_Class {
    public:
        char* name (); // Class name
           // class structure (data members)
           // for (un)marshalling of objects, etc.
        int mb_nb ();                    // number of data members
        char*    mb_name (int rank);     // names
        EC_Class mb_type (int rank);     // type
        EC_Any   mb_get (int rank, EC_Any obj);             // get a member
        void     mb_set (int rank,EC_Any obj,EC_Any value); // set a member
```

```
 size_t   object_size ();        // size of instance object

    // class member functions
 int mbf_nb ();                  // number of member functions
 EC_Mb_Fct mbf_get (int rank);   // get a member function representation

    // Object instantiation
 EC_Any new_obj (EC_Call* c);
       // Create a new object of the current class, calls the
       // constructor specified by EC_Call,
       // default constructor if c is null
 EC_Any new_obj ();
       // Same without constructor being called, allocation only
 EC_Any new_reified_obj (EC_Proxy* pr);
       // Create, for the current class, a reified object with the
       // proxy object pr. Returns null if class cannot be reified
       // The actual object itself is not created since it is
       // the proxy responsability.
 bool is_reifiable ();
       // Is a class that can be reified?

   // Classification of types:
       // Inheritance
bool is_derived_from (EC_Class c); // Is a kind of?
       // RTTI information
bool is_polymorphic(); // Is a polymorphic type?

       // Qualifiers
bool is_const();         // Is a const object?
bool is_volatile();       // Is a volatile object?
bool is_const_volatile(); // Is a const volatile object?

       // Is a fundamental types?
EC_Type  is_ftype (); // If not returns 0,
                      // otherwise code for use with enum EC_Type

       // Compound types
bool is_array ();      // Is a C++ array?
bool is_fct_ptr ();    // Is a function pointer?
bool is_mb_fct_ptr (); // Is a member function pointer?
bool is_charst ();     // Is a char*?
bool is_struct ();     // Is a struct?
bool is_union ();      // Is an union?
bool is_obj ();        // Is a class (not a ref, not a pointer)?
bool is_ref ();        // Is a reference?
bool is_ptr ();        // Is a pointer?

       // Special cases for EUROPA EC_Any
bool is_ecany ();      // An EC_Any object represented by an EC_Any?
```

```
            // Manipulation of pointer or reference types,
            // returns  a non vallid EC_Class if operation is illegal
      EC_Class deref  ();     // dereference a pointer type
      EC_Class derefs ();     // upto a non pointer type
      EC_Class addptr ();     // take the pointer type
      EC_Class addref ();     // take the reference type
      bool is_illegal ();     // Is current EC_Class object not a valid type?

            // Equality, inequality
      bool operator == (EC_Class c);
      bool operator != (EC_Class c);

    protected:
            // Implementation defined
    };
}
```

The following enum gives constant *Class_id* values to fundamental C++ types:

```
    namespace EC {
        enum EC_Type {
            ec_char,    // char
            ec_uchar,   // unsigned char
            ec_schar,   // signed char
            ec_enum,    // enumeration
            ec_short,   // short integer
            ec_int,     // integer
            ec_lgint,   // long integer
            ec_ushort,  // unsigned short int
            ec_uint,    // unsigned int
            ec_ulong,   // unsigned long int
            ec_wchar_t, // extended character
            ec_bool,    // boolean
            ec_float,   // float
            ec_double,  // double
            ec_lgdouble,// long double
            ec_void,    // void type
        }
    }
```

#### 4.2.3 Member function representation: EC_Mb_Fct

The *EC_Mb_Fct* class provides identification for member functions, as well as the ability to call a member on an object, with effective parameters.

```
namespace EC {
    class EC_Mb_Fct {
    public:
        char* name(); // Member function name
```

```
        EC_Class return_type (); // Return type
        int formal_nb();         // Number of formals parameters
        EC_Class* formals();     // List of formal parameter types
        EC_Any execute (EC_Any obj, EC_Any* eff);
              // Execute the member, using the effective parameters, and
              // returns the result. Overriding resolution (Dynamic Binding),
              // but no overloading resolution here.
        EC_Class decl_class ();  // Declaration class of the member

        EC_Call* new_call(EC_Any* eff);
              // Returns a call object for the current member function
              // with effective parameters.

        bool is_static (); // Is the mb. fct. static?
    };
}
```

The member function is a constructor when its name is equal to its class name (*name()==decl_class().name()*), and similarly for destructor with the extra character $\sim$.

*EC_Class* and *EC_Mb_Fct* classes are *meta-classes* that represent information on C++ classes. They conform to the design principle of the standard class *type_info* which provides some basic information, and are designed with the intention of being extended according to specific needs.

By specification, *EC_Class* and *EC_Mb_Fct* values must have a system wide validity (they can be passed as parameters between processes).

#### 4.2.4 Mapping and conversion primitives

Primitives to obtain class and member identifications (*EC_Class*, *EC_Mb_Fct*), and conversion to RTTI *type_info* references are provided. *type_info* is the standard RTTI class of C++.

```
namespace EC {
               // Class identification: ec_cid
        EC_Class ec_cid (typename);     // ec_cid(A)

               // Member identification: ec_mid
        EC_Mb_Fct ec_mid1 (membername);           // mid(put)
        EC_Mb_Fct ec_mid2 (typename, membername); // ec_mid(A,put)
        EC_Mb_Fct ec_mid3 (typename, membername, typename);
                                                  // ec_mid(A,put,T1)
        EC_Mb_Fct ec_mid4 (typename, membername, typename, typename);
                                                  // ec_mid(A,put,T1,T2)
        ...
        EC_Mb_Fct ec_mid12 ( ... ); // ec_mid(A,put,T1,T2, ..., T10)

               // Conversion from EC class id to RTTI type_info
        type_info& ec2rtti (EC_Class c);
```

```
        EC_Class   rtti2ec (type_info& t);

}
```

### 4.2.5 Generation of information

The EC++ standard specifies that each class or type with its name appearing within a *typeid* primitive, or being derived from *EC_Reflect* must be a reifiable class, and also have the corresponding *EC_Class* and *EC_Mb_Fct* classes to be generated and made available at runtime.

Besides this implicit generation, the EUROPA standard provides a means to control the amount of information to be generated. An important feature is the ability to define the strategy of generation on classes that *do not exist yet*. The intent is to give the programmer of a library for parallel programming (Level 1) the ability to control the amount of information generated for classes that he does not know about: the application programmer classes. Level 0 provides the access to such a crucial mechanism.

The information to be generated by an EUROPA compliant pre-processor can be classified into three categories for a given class $A$:

1. The ability to reify $A$ (called *reifiable*).
2. The ability to access dynamically the member functions of $A$ (*member functions*); corresponds to the generation of a set of *EC_Mb_Fct* instances.
3. The ability to access dynamically the class representation of $A$ (*class structure*); corresponds to the generation of a single *EC_Class* class instance.

For consistency, those three degrees are classified in the following manner:

*reifiable* => *member functions* => *class information*

meaning that if $A$ is reifiable, then it also has the generation of its member functions, and in that case it also implies that its class structure exists at runtime.

Although theoretically those three degrees could be controlled independently, the current definition of the standard, for the sake of simplicity, collapses *member functions* and *class structure* controls into a single one.

As a consequence, Level 1 programmers can control the amount of information generated through two classes: *EC_Reifiable* and *EC_Generation*. All the calls to their static functions have to occur within classes that are derived from *EC_Reflect*.

```
namespace EC {
    class EC_Reifiable { // Controls classes that are reifiable at runtime
    public:
        static void single (type_info& t);
            // Generates reifiable for a given class
            // Usage: EC_Reifiable::single (typeid(A));
```

```
        static void base (type_info& t);     // All base classes
        static void derived (type_info& t);  // All derived classes
        static void hierarchy (type_info& t);// Base and derived

        static void data_members (type_info& t);
            // Data member types of a given class
        static void all_data_members (type_info& t);
            // Recursively all the data members of a given class
            // (transitive closure)

        static void parameters (type_info& t);
            // Parameter types of member functions within hierarchy
        static void return_types (type_info& t);
            // Return types of member functions within hierarchy

        static void related (type_info& t);
            // Classes related to a given class: based, derived,
            // data members, parameters, and return types of
            // member functions
        static void all_related (type_info& t);
            // All classes related: transitive closures

        static void all_related_lim (type_info& t,type_info& lim1,...);
            // Idem to all_related, but do not follow on the transitive
            // closures after limit types given in ellipsis parameters
        static void all_data_members_lim(type_info& t,type_info& lim1,...);
            // Idem to all_data_members with limits

        static void all ();
            // For all classes in the system
    };
}

namespace EC {
    class EC_Generation {
       // Controls the generation of EC_Mb_Fct and EC_Class classes
    public:
        static void single (type_info& t);
            // Generates EC_Mb_Fct and EC_Class for a given class
            // Usage: EC_Generation::single (typeid(A));

        static void base (type_info& t);     // All base classes
        static void derived (type_info& t);  // All derived classes
        static void hierarchy (type_info& t);// Base and derived

        static void data_members (type_info& t);
            // Data member types of a given class
        static void all_data_members (type_info& t);
            // Recursively all the data members of a given class
```

```
            // (transitive closure)

        static void parameters (type_info& t);
            // Parameters types of member functions within hierarchy
        static void return_types (type_info& t);
            // Return types of member functions within hierarchy

        static void related (type_info& t);
            // Classes related to a given class: based, derived,
            // data member, parameters, and return types of
            // member functions
        static void all_related (type_info& t);
            // All classes related: transitive closures

        static void all_related_lim (type_info& t,type_info& lim1,...);
            // Idem to all_related, but do not follow on the transitive
            // closures after limit types given in ellipsis parameters
        static void all_data_members_lim(type_info& t,type_info& lim1,...);
            // Idem to all_data_members with limits

        static void all ();
            // For all classes in the system
    };
}
```

# 5 Examples

This section presents a few examples to illustrate how library programmers can build various paradigms for parallel and distributed computing within EUROPA. Moreover, it demonstrate how those libraries can be used in a very simple and transparent manner by application programmers.

## 5.1 A first example

This first simple example of the use of the EUROPA standard just builds a class that prints out the name of each public member called. It also uses an object (of type *Color*) which represents some allocation strategy for the reified object; it simulates some mapping strategy, e.g. in the case of the creation of a remote object.

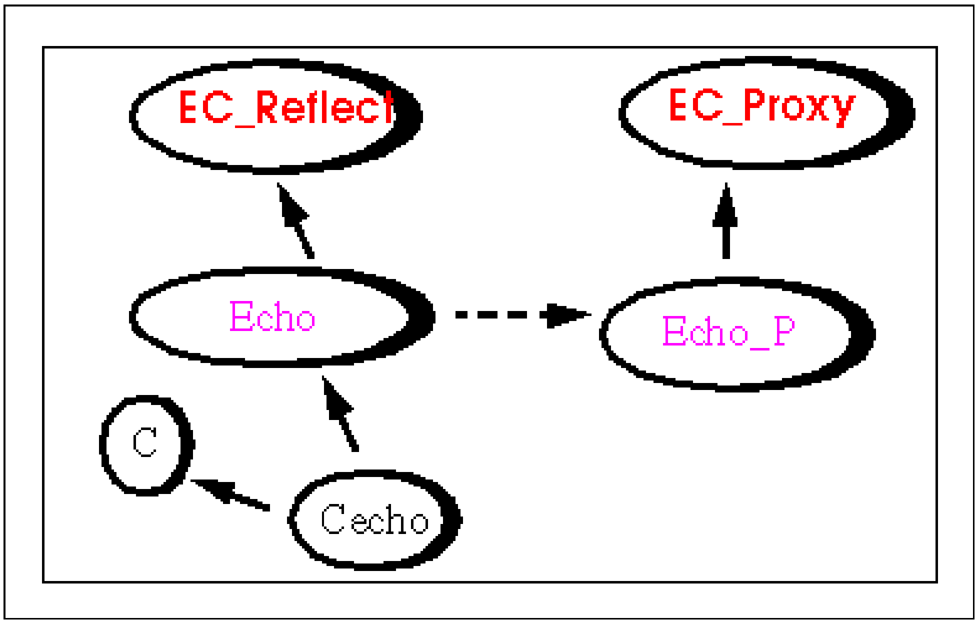


Figure 7: Programming the echo behavior

### 5.1.1 Reified class

The first thing is to declare a reified class, inheriting from *EC_Reflect*; the user will just inherit from that class in order to get the desired behavior.

```
namespace EC {
   class Echo: public EC_Reflect {
     void* operator new (size_t s, Color m);
```

```
  protected:
    Echo_P* ec_proxy();
  };
}
```

The *Echo_P* class, a proxy class, implements the printing behavior. The allocator new allows for passing the mapping parameter.

Now, all classes that will inherit from *Echo* will be automatically given the corresponding proxy, and as such, its behavior.

### 5.1.2 Proxy definition

The second thing to define, as a library programmer, is the proxy class that implements the behavior:

```
namespace EC {
class Echo_P : public EC_Proxy {

  EC_Any obj;   // Reified object
  Color mapping; // Some allocation or mapping value

  void ec_reify (EC_Call* c) {
    cout << c->mb->name() << endl; // print out the name
    c->res=c->execute(obj); // do the call
  }

  // Constructor
  Echo_P (Color m, EC_Class cl, EC_Call* c) {
    mapping = m;
    obj = cl.new_obj(c); // Create the reified object
  }
};
}
```

The class holds a reference to the object for which it acts as a proxy (*obj*). The attribute *mapping* (type *Color*) holds the mapping value. The routine *ec_reify* programs the desired behavior:

- prints out the name, and then
- actually executes the call with no modifications.

Finally, the constructor routine receives as a first parameter the allocator parameter (of type *Color*, not really used in that example), and then the type of the actual object being created (*EC_Call*), and the reification of the constructor (*EC_Class*). The routine just creates the object, and assigns it to the *obj* attribute of the proxy in order to keep a reference to it.

### 5.1.3 Using the reified class

Let us say we have a typical user class of the form:

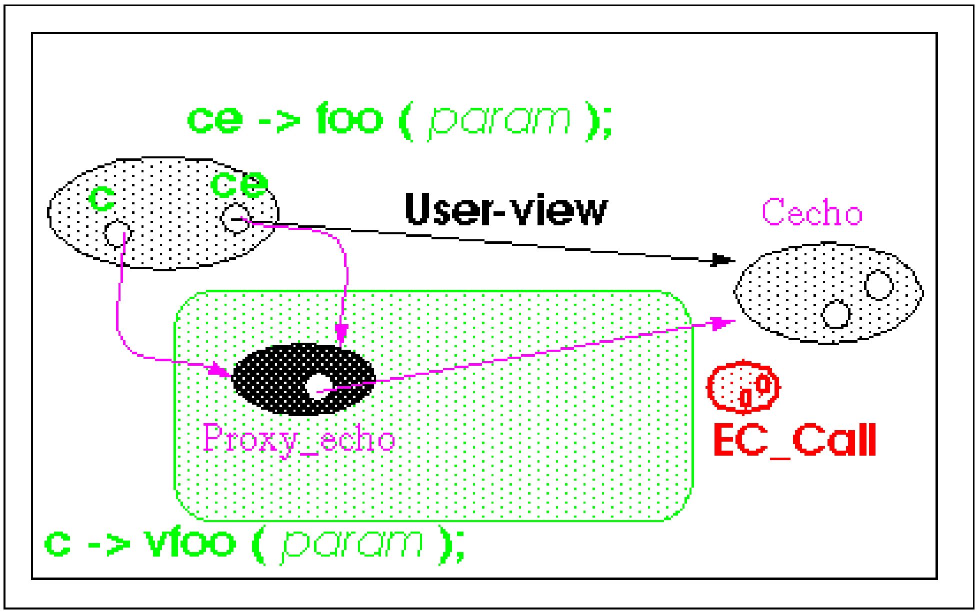


Figure 8: Runtime view of the echo behavior

```
class C { // Any user class
public:
  Res* foo ( param );
  virtual Res* vfoo ( param );
  // Constructors
  C(P1 p1);
  C(P1 p2);
};
```

If a final user needs to add the echo behavior to this class, he will only need to declare the following:

```
class Cecho : public C, public Echo {
public:
  Cecho(P1 p1): C(p1)
  Cecho(P1 p2): C(p2)
};
```

Then the class *Cecho* can be used directly:

```
void main() {
  P1 p1;
  Color green;

  Cecho* ce = ec_new ((green), Cecho, (p1));
```

```
  r = ce->vfoo(...);
  r = ce->foo (...);
}
```

Both functions will present the echo behavior. If one needs to use the *Cecho* class in an existing application, it can be done through polymorphism between entities of type *C* and *Cecho*:

```
void main() {
  P1 p1;
  Color green;

  C* c;
  Cecho* ce = ec_new ((green), Cecho, (p1));

  c = ce; // Polymorphic assignment
  ⋮
  r = c->vfoo(...);
}
```

Of course in this case, due to the dynamic binding that occurs only on virtual functions in C++, the echo behavior will occur only on those functions declared virtual in *C*.

## 5.2 Asynchronous calls and transparent futures

This second and more sophisticated example of EUROPA standard demonstrates that it is actually possible to build a library class that achieves asynchronous calls with a transparent futures semantics on the results of those calls.

In order to obtain the asynchronous execution of member functions thanks to threads, the library user will just inherit from the reified class *AThreaded*:

```
namespace EC {
   class AThreaded: public EC_Reflect {
   public:
     Future_P* get_future (EC_Any obj);
   };
}
```

All the calls made on instances of *AThreaded* derived classes will be executed asynchronously and potentially concurrently by a pool of threads.

A special function, *get_future*, applied to the result of an asynchronous call, returns a future accessor (a proxy) that provides for explicitly testing its status; its returns null when apply on other objects. The *Future_P* interface is defined as follows:

```
namespace EC {
   class Future_P {
   public:
     void   Wait();    // Wait until the future is known
```

```
    bool   Awaited(); // Test if future is known
    EC_Any value();   // Wait, and returns the value
  };
}
```

Using the class *C* of the *echo* example, the class *AThreaded* can be used in the following way:

```
  class AC: public C, public AThreaded {
  public:
    AC(P1 p1): C(p1) {}
    AC(P1 p2): C(p2) {}
  };
```

The class *AC* now features asynchronous calls in a transparent manner:

```
void main() {
  P1 p1;
  Res* v1,v2;
  AC* ac = ec_new ((), AC,(p1));
  v1 = ac->vfoo(...); // Asynchronous call
  v2 = ac->foo (...); // Asynchronous call
  v1->bar(); // Automatically triggers a wait until the
  v2->bar(); // value is actually returned from a thread
     // The only constraint being that function bar has to be virtual

  C* c;
  c = ac; // Polymorphic assignment
  ...
  v1 = c->vfoo(...); // Asynchronous call
}
```

If the user needs to check the results availibility, to avoid being blocked on an unknown future, he can use the following scheme:

```
void main() {
  AC* ac;
  v1 = ac->vfoo(...); // Asynchronous call
  v2 = ac->foo (...); // Asynchronous call

  Future_P* f1=ac->get_future (EC_Any(v1));
  Future_P* f2=ac->get_future (EC_Any(v2));

  while (f1->Awaited() && f2->Awaited() )
     ... do something else ...
}
```

### 5.2.1 Threads

We now present the implementation of such a library. First, the class *AThreaded* is defined as follows:

```
namespace EC {
   class AThreaded: public EC_Reflect {
   private:
     AThreaded_P* ec_proxy();
   public:
     Future_P* get_future (EC_Any obj) {
       EC_Proxy* p = ec_proxy(); // Needs to use ec_proxy(), such that the
                                 // current function is not reified
       EC_Proxy* pr = obj.is_reified();
       if (pr)
          return (dynamic_cast<Future_P*> (pr));
       else
          ... error or build a proxy anyway
     }
   };
}
```

The proxy itself being imlemented by:

```
namespace EC {
   class AThreaded_P: public EC_Proxy {
   public:
     void ec_reify (EC_Call* c) {
        EC_Class rt = c->mb()->return_type();
           // Get result type of mb fct being called
        Future_P* fp = new Future_P()   // Create a future proxy
        EC_Any fut = rt->new_reified_object(fp);
           // Creates a reified object of the return type with a
           // future proxy behavior
        t->enqueue (c,obj,fut);
           // Add the call, the object, and the future to the list
           //  of pending calls to be executed by the threads
        c->res = fut; // Returns the future
     }
   // Constructor
     AThreaded_P(EC_Class cl, EC_Call* c){
        obj = cl->new_obj(c);
            // Create the object to be called asynchronously
        t = ... get threads one way or another ...
     }
   protected:
     EC_Any obj;     // Reified object: called asynchronously
     My_threads t;   // Pool of threads
   };
}
```

The *ec_reify* function constitutes the new and the most interesting part of this definition. Each time member function is called, a reified object of the result type is created with a future behavior. Then, the future together with the reified call are enqueued in the list of pending calls; the result is set as the future.

Later on, the call will be executed by one thread with the following code:

```
 ... get a pending call c, the object obj, and its future fut ...
 fut->set_value(c->execute(obj));
```

At this point, an actual implementation would have some extra mechanism to resume the waiting objects — it could be implemented within the *set_value* member.

### 5.2.2 Futures

The classes that program the future behavior are the following. First, we have to program the reified class that allows to get the future behavior from standard classes. This class was used in the *AThreaded_P* proxy class in order to build a future for any user return type (*new_reified_object(ec_cid(Future))*).

```
namespace EC {
   class Future: public EC_Reflect {
      virtual void Wait() {} // Reified
      virtual bool Awaited() { // Not Reified
          return (ec_proxy()->Awaited());
      }
   protected:
      Future_P* ec_proxy(); // Future proxy
   };
}
```

In that case, we can notice that the reified class *Future* has two member functions that are specific to such a class: *Wait* and *Awaited*. Defining the functions in the reified class allows to add them to classes that will be defined inheriting from *Future*. Note that *Awaited* accesses the proxy, and as such is not reified, while *Wait* does and as a consequence is reified (this is the simpliest way to define it, but for efficiency purpose it could be defined without reification by *ec_proxy()->Wait()*).

The proxy future class that implements the future behavior is as follows:

```
namespace EC {
   class Future_P: public EC_Proxy {
   public:
     void ec_reify (EC_Call* c) {
       Wait(); // Wait until the future is known
       c->res=c->execute(obj); // Do the call
     }
     void Wait() { if (obj==0) wait_obj(); }
     bool Awaited () { return(obj!=0); }
     EC_Any value (){ Wait(); return (obj); }
     void set_value (EC_Any r) { obj=r; }
   // Constructor
     Future_P (){
       obj=0; // Set the object to unknown
     }
```

```
  protected:
    EC_Any obj;   // Reified object to be accessed through the future
  };
}
```

### 5.3 Active remote objects, asynchronous calls, and transparent futures

This third example of EUROPA standard illustrates how to build, as a library, active objects with a remote instantiation, e.g. on a cluster of machines. The framework also achieves asynchronous calls with transparent futures semantics on the results of those calls, and automatic (un)marshalling of parameters.

This particular library implements a MIMD model [Car93] with:

- sequential processes,
- asynchronous communications,
- wait-by-necessity [Car89],
- and without shared objects.

The wait-by-necessity (automatic and transparent futures) allows the parallelization of object-oriented sequential system in a *data-flow* manner.

The programming of this Level 1 library is based on the introduction of a *Process* class.

All objects which are an instance of a class which publicly inherits from the *Process* class are processes (active object = process). Passive objects (i.e. objects which are not active) belong at run-time to a process object, thus giving a distributed structure of active objects encapsulating passive objects.

A communication between active objects is simply programmed as an asynchronous member function call:

```
p->foo( parameters );
res = p->bar ( parameters );
```

Such a function calls between active objects (processes) is implicitly (by default) asynchronous, thus allowing and encouraging parallel execution of objects (although a synchronous method call is also supplied but must be stated explicitly in the function call itself or in the process definition). The wait-by-necessity adds automatically the synchronizations when needed.

The semantics of communication between processes is a copy semantics for passive objects: all parameters are automatically transmitted by copy from one process to another. Of course, active objects are subject to a reference semantics: all processes are transmitted by reference from one process to another. As a consequence, shared objects (i.e. passive objects which are shared between active objects) are not allowed as such. The implication of this is that any passive object which is to be shared between several active objects must be made an instance of a class derived from process. This is simply achieved by inheriting from the process class (which also provides an automatic FIFO synchronization).

### 5.3.1 Process

Let us start this time with the reified class *Process* that represents a *process class* from which library users inherit in order to instanciate remote active objects.

```
namespace EC {
   class Process: public EC_Reflect, public Request_line {
       // A reified class.
       // Inheriting from Request_line brings all the service routines
       // and the pending request management.
   public:
      void* operator new (size_t s, Mapping m);
        // Mapping allows to control on what machine the allocation
        // takes place
      void Live () {
         // Behavior of the active object: default is a FIFO service
         // Called by the Process constructor
         EC_Generation::all_related(typeid(Process));
         EC_Reifiable::return_types(typeid(Process));
         ...
      }
      Process () { // Constructor
         Live ();
      }
   protected:
      Process_P* ec_proxy();
   };
}
```

### 5.3.2 Using remote, asynchronous, active objects

Using the class $C$ of the *echo* example, the user of the library can get asynchronous remote $C$ as follows:

```
   class ARC: public C, public  Process{
   public:
     ARC(P1 p1): C(p1), Process () {}
     ARC(P1 p2): C(p2), Process () {}
   };
```

The class $ARC$ now features remote instantiation, and asynchronous calls on its member function in a transparent manner:

```
void main() {
  P1 p1;
  Res* v1,v2,v3;
  Mapping map;
```

```
  ARC* arc1 = ec_new ((map(``Inria.Sophia.fr'')), ARC,(p1));
  ARC* arc2 = ec_new ((map(``Nice.Sophia.fr'')), ARC,(p1));

  v1 = arc1->vfoo(...); // Asynchronous remote calls
  v2 = arc2->vfoo(...);
  v3 = arc1->foo (...);

  v1->bar(); // Automatically triggers a wait until the
  v2->bar(); // value is actually returned from a thread
     // The only constraint being that function bar has to be virtual

  arc1->set_neighbor(arc2);
             // Remote objects can be pass as parameters to remote
             // calls, providing dynamic topology.

  C* c;
  c = arc1; // For the sake of reuse, polymorphic assignment is
            // possible between local entities and remote objects
  ...
  v1 = c->vfoo(...); // Asynchronous remote call
}
```

Of course, in that case again the user can check the results availibility with the technique presented for the previous example.

### 5.3.3 Programming the library

In order to program this library, the proxy for *Process* is defined as follows:

```
namespace EC {
class Process_P: public EC_Proxy {
public:
  // Constructor
  Process_P (Mapping map, EC_Class cl, EC_Call* c) {
    robj=new_robj_id (map);
               // Interpret the mapping value,
               // and returns a remote object id: obj ID, Machine ID
    remote_create (robj,cl,c);
               // Actual remote creation of the reified object
               // of type cl, constructor represented by c will
               // be called.
               // Usually executed synchronously
  }

  // Call reification
  void ec_reify (EC_Call* c) {
       // Get a new request id
    req_id= new_req_id();
       // Future creation
    EC_Class rt = c->mb()->return_type(); // Get result type
```

```
    Future_P* pr = new Future_P(); // Build a future proxy
    EC_Any fut = rt->new_reified_object(pr); // reified future object
       // Keep track of the correspondance between future and
       // request id for later update when result will be received
    enqueue (fut,req_id);
       // Remote non-bloking call
    remote_send (robj,req_id,flat(c));
       // The reified call is sent to the remote object.
       // Effective parameters held in c are marshaled by the flat
       // function: call flat on the object if user-defined, otherwise
       // calls a generic flat.
       // The result will come back together with the request id,
       // such that the corresponding future will be updated with the result
  }
protected:
  Remote robj;   // Remote object reference: network and transport aware
};
}
```

The classes *Future* and *Future_ P* of the previous example are reused with no change.

## 5.4 Simulation objects

In some applications, the user of a reified object – the object calling a reified function – needs to be known or accessed within the proxy. This is a quite well known and commonly required feature that many meta-object protocols have problems achieving.

Within the EUROPA standard, this functionality can be obtained in a quite straighforward manner, as explained in the rest of this section. The solution relies on the ability to define member functions within reified classes are not reified, but executed directly by the proxy.

To illustrate this feature, we will take the example of a library offering Parallel Discrete Event Simulation based on active objects, following the design presented in [BBC+96]. A simulation is a collection of concurrently *active simulation objects* derived from a special class, let us say `Sim_Process`. Such classes have public member functions that represent the services being simulated.

Typically, such a library can be defined using EUROPA, first taking advantage of reification to control the execution of services within the simulation time, but also to achieve the parallel/distributed execution.

One reason that leads to the need of the special functionality – the object calling a reified function being accessed within the proxy – is two fold:

- some control parameters need to be added to the call (e.g. the priority of the call, the maximum simulation time allowed for the service),
- those parameters might need to be evaluated within the proxy according to the object making the call (`this`).

Let us say we have a simulation object *Sim_ Supermarket*, that represents a supermarket simulation, holding a service routine *enter* modelling the entrance of a new client.

The simulation class can feature a special routine, *setup*, defined on the `Sim_Process` as a non reified routine to be executed directly by the proxy — using a technique similar to the *Awaited* routine defined in the future example (class *Future*, section 5.2.2). Then, the simulation object can be used as follows:

```
void main() {
   Sim_Supermarket* sim_obj;
   sim_obj = ec_new ((), Sim_Supermarket, ());

   sim_obj->setup(this,max_prority,time(10)); // Not reified
   sim_obj->enter(blonde,lady,thin);           // Reified
   ...
   sim_obj->setup(this,min_priority,time(10));
   sim_obj->enter(blond,guy,tall);
}
```

If one really requires to have both the call to the original function, and the extra parmeters within a single and somehow atomic call, a macro can be used to hide the two calls being executed:

```
#define SIM_CALL(object , call , setup1, setup2, setup3)
          (object->setup(setup1,setup2,setup3), object->call)

   SIM_CALL(sim_obj,enter(blonde,lady,thin), this,max_priority,time(10));
   SIM_CALL(sim_obj,enter(blond,guy,tall), this,min_priority,time(10));
}
```

Please note that this technique provides a means to make the call atomic in a convenient manner as locking functions can be included within the macro. The technique is also valid when the service being called needs to return a value:

```
val=SIM_CALL(sim_obj,leave(blonde,lady,thin),this,max_priority,time(10));
```

Another possibility, if a reified object is used by a unique client, is to set the user object at construction time. Then, the parameter *this* does not need to be specified for each call any more, and the control parameters can be set directly within the client, whenever they need to be modified; the proxy has the possibility to access them directly.

```
void main() {
   Sim_Supermarket* sim_obj;
   sim_obj = ec_new ((), Sim_Supermarket, (this));

   prority = max_prority; time = 10;
   sim_obj->enter(blonde,lady,thin);
   ...
   prority = min_priority
   sim_obj->enter(blond,guy,tall);
}
```

## 5.5 Attaching properties to member functions

In some cases, the user of a library (reified) class would like to be able to attach properties to member functions. For instance, carrying on with the supermarket example, one might want to specify that some services are called asynchronously, while others synchronously.

Again, a routine defined on the `Sim_Process` as a non reified routine to be executed by the proxy allows this. With such a routine called *set_mb_semantics*, the scheme is the following:

```
void main() {
   Sim_Supermarket* sim_obj;
   sim_obj = ec_new ((), Sim_Supermarket, ());

      // Initialization
   sim_obj->set_mb_semantics (mid2(Sim_Supermarket,enter), ASYNC);
   sim_obj->set_mb_semantics (mid2(Sim_Supermarket,leave), SYNC);

      // Use
   sim_obj->enter(blonde,lady,thin);           // Asynchronous call
   ...
   time_used=sim_obj->leave(blonde,lady,thin); // Synchronous call
}
```

Of course, since the set up of the properties is achieved dynamically, the user might also change it during the execution if needed.

## 5.6 Attaching properties to parameters

In a similar trend, but going a little bit further than the previous example, one might need to attach properties to arguments of member functions. For instance, in a distributed implementation, the user of a remote object could specify that some parameters are transmitted with a deep copy semantics, while others only require a reference semantics.

In that case again, a non reified routine (*set_param_semantics*) can provide for such pattern:

```
void main() {
   Sim_Supermarket* sim_obj;
   sim_obj = ec_new ((), Sim_Supermarket, ());
      // Initialization
   sim_obj->set_param_semantics (mid2(Sim_Supermarket,enter),COPY,REF,COPY);
      // Use
   sim_obj->enter(blonde,lady,thin); // blonde and thin have copy semantics
                                     // lady a reference semantics
   ...
}
```

# 6 Conclusion

To summarize, EUROPA does not define a parallel C++ language per se. Instead it defines a standard structure allowing the programming of various parallel programming models within C++.

The standard gives a basic structure to which standard parallel C++ environments can adhere, thus making portable and reusable code a more attainable goal.

Eventually, EUROPA libraries will include an interface to the CORBA (Common Object Request Broker Architecture) of the Object Management Group (OMG). The proxy model will permit to automatically generate object description, and thus will allow for transparent access to a CORBA Object Broker.

The definition of the EUROPA standard has been done in such a way as to allow the implementations to be targeted at a wide variety of parallel architectures, and for any architecture specific optimizations and requirements. In the same way the standard has been defined so as to permit as many parallel programming models and paradigms as possible to fit in the standard — indeed the level 1 of the standard is dedicated to the definition of parallel models of computation within C++.

In this way EUROPA offers a standard and portable development environment within which the development of parallel C++ and applications can be achieved.

# References

[ABB+91] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. PANDA—Supporting Distributed Programming in C++. In O.M. Nierstrasz, editor, *ECOOP'93—Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 361–383. Springer-Verlag, 1991.

[ACO93] I. Attali, D. Caromel, and M. Oudshoorn. A Formal Definition of the Dynamic Semantics of the Eiffel Language. In G. Gupta, G. Mohay, and R. Topor, editors, *Sixteenth Australian Computer Science Conference (ACSC-16)*, pages 109–120. Griffith University, February 1993.

[Ada91] J.-M. Adamo. Extending C++ with Communicating Sequential Processes. In P. Welch, D. Stiles, T. Kunii, and A. Bakkers, editors, *Proceedings of Transputing'91*, volume 1, pages 431–448. ISO Press, 1991. ISBN 90-5199-045-6.

[Agh90] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[Ame90] P. America. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *Proceedings of ECOOP/OOPSLA '90*, pages 161–168, 1990.

[BBC+96] Françoise Baude, Fabrice Belloncle, Denis Caromel, Nathalie Furmento, Philippe Mussi, Yves Roudier, and Günther Siegel. Parallel Object-Oriented Programming for Parallel Simulations. *Information Sciences Journal, Informatics and Computer Science*, 93(1-2):35–64, 1996. Special issue on Object-Oriented Programming.

[BBG+93] F. Bodin, P. Beckman, D. Gannon, S. Yang, K. Shanmugam, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of Supercomputing '93*, pages 588–597, November 1993.

[Car89] D. Caromel. Service, Asynchrony and Wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, November 1989.

[Car90a] D. Caromel. Concurrency: an Object Oriented Approach. In J. Bezivin, B. Meyer, and J.-M. Nerson, editors, *Technology of Object-Oriented Languages and Systems (TOOLS'90)*, pages 183–197. Angkor, June 1990.

[Car90b] D. Caromel. Concurrency and Reusability: From Sequential to Parallel. *Journal of Object-Oriented Programming*, 3(3):34–42, September 1990.

[Car93] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[CGH94] R. Chandra, A. Gupta, and J.L. Hennessy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, 27(8):14–26, August 1994.

[CK92] K.M. Chandy and C. Kesselman. The CC++ Language Definition. Technical Report Caltech-CS-TR-92-02, California Institute of Technology, 1992.

[CK93] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-oriented Programming Notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press, 1993. ISBN 0-272-01139-5.

[Con95] Concurrent Systems Architecture Group. The ICC++ Reference Manual. Technical report, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, Illinois, 1995. Also available from http://www-csag.cs.uiuc.edu/.

[CR95] A.A. Chien and U. Reddy. ICC++ Language Definition. Concurrent Systems Architecture Group Memo, February 1995.

[GBD+94] A. Geist, A. Beguelin, J.J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM (Parallel Virtual Machine): A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[GR88] N.H. Gehani and W.D. Roome. Concurrent C++: Concurrent Programming with Classes. *Software—Practice and Experience*, 18(12):1157–1177, 1988.

[Gri93] A.S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[KH93] D.J. Kickly and P.J. Hatcher. C++ and Massively Parallel Computers. In *Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 271–283, April 1993.

[KK93] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, 1993.

[Lar93] J.R. Larus. C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages And Compilers for Parallel Computing (5th International Workshop)*, pages 326–341. Springer-Verlag, August 1993.

[LG91] J.K. Lee and D. Gannon. Object Oriented Parallel Programming: Experiments and Results. In *Proceedings of Supercomputing '91*. IEEE Computer Society and ACM SIGARCH, 1991.

[MMB+94] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers. In H.J. Siegel, editor, *Proc. Eighth International Parallel Processing Symposium*. IEEE Computer Society Press, April 1994.

[MPI94] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[ORW+95] T. O'Brien, R. Roberts, M. Wei, R. Winder, D. Titcombe, M. Plaxton, A. McEwan, and J. Poole. UC++ V1.4 Language and Compiler Documentation. Technical Report TR/95/8, Department of Computer Science, University College London, 1995.

[RFH+ar] J.V.W. Reynders, D.W. Forslund, P.J. Hinker, M. Tholburn, D.G. Kilman, and W.F. Humphrey. OOPS: An Object-Oriented Particle Simulation Class Library for Distributed Architectures. *Computational Physics Communications*, to appear.

[RW90] G. Roberts and R. Winder. Integrating Symbolic and Numeric Computation. Research Note RN/90/40, Department of Computer Science, University College London, 1990.

[RW94] G. Roberts and R. Winder. UC++: Parallel and Distributed Programming with C++. Research Note RN/94/34, Department of Computer Science, University College London, 1994. Poster Paper presented at OOPSLA'94.

[RWW90] G. Roberts, R. Winder, and M. Wei. UC++: Concurrent Object Oriented C++. In *Proceedings of the 2nd Workshop on Parallel Object Oriented Programming*. BCS Object-Oriented Programming Systems and BCS Parallel Processing Specialist Groups, 1990.

[RWW91] G. Roberts, M. Wei, and R. Winder. COOTS Project Subtask 1.1 Deliverable – UC++ and CoSIDE Design and Implementation. Technical Report TR/92/8, Department of Computer Science, University College London, 1991.

[RWW92] G. Roberts, M. Wei, and R. Winder. COOTS Deliverable 1.2—UC++ compiler and CoSIDE. Technical Report TR/92/30, Department of Computer Science, University College London, 1992.

[SC90] K. Smith and A. Chatterjee. A C++ Environment for Distributed Application Execution. Technical Report ACT-ESP-015-91, Microelectronics and Computer Technology Corporation (MCC), November 1990.

[Sel90] R. Seliger. Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling and Garbage Collection. In *Proceedings of Usenix C++ Conference*, pages 241–264, 1990.

[Str91] B. Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991.

[WJK92] R. Winder, G. Joly, and A. Kamalati. Applications in CoSIDE. *OOPS Messenger*, 4(2):215–216, 1992.

[WRW92] R. Winder, G. Roberts, and M. Wei. CoSIDE and Parallel Object Oriented Languages. *OOPS Messenger*, 4(2):211–213, 1992. Poster paper presented at OOPSLA'92.

[WWR91] R. Winder, M. Wei, and G. Roberts. Harnessing Parallelism with UC++. In *Proceedings of ECUG'91*, pages 101–114. European C++ User Group Conference, 1991.

[WWR92] R. Winder, M. Wei, and G. Roberts. UC++: An Active Object Model for Parallel C++. Research Note RN/92/115, Department of Computer Science, University College London, 1992.

[WWR93a] M. Wei, R. Winder, and G. Roberts. Introducing UC++—A natural progression from C, through C++, and on to parallel C++. Internal Note IN/93/7, Department of Computer Science, University College London, 1993.

[WWR93b] M. Wei, R. Winder, and G. Roberts. UC++ Definition. Internal Note IN/93/8, Department of Computer Science, University College London, 1993.