Esprit Project N^0 6603 OUVERTURE

An Overview of COOL and COOL–IDL

date:	March 1994	*ref:*	**OU/TR-94-3**
category:	Documentation	*workpackage / task:*	T1.1
distribution:	partners		
clearance:	Esprit	*status:*	Final
authorised by:	Christian Jacquemot	*classification:*	Public

author(s):

C. Jacquemot
F. Herrmann
P. S. Jensen
Chorus systèmes

P. Gautron
J. Mukerji
Novell/USG

H.G. Baumgarten
H. Hartlage
Siemens AG

abstract: This paper is a collection of two papers (a technical paper and a position paper) on the CHORUS approach to the OMG specification.

Contents

1 Introduction **4**

2 COOL–IDL **5**

2.1 COOL–IDL Language 5

2.2 COOL–IDL Programming Model 6

2.3 COOL–IDL Syntax and Semantics 7

3 Generic COOL Architecture **9**

3.1 Stub and Skeleton Classes 10

3.2 Optimizing Co-location Invocations 10

3.3 Cluster Class 11

3.4 Invocation Classes 12

3.5 Memory Management Classes 12

4 The Family of COOL Supports **12**

4.1 COOL–L Architecture 14

4.2 COOL–LD Architecture 15

4.3 COOL–LDMP Architecture 16

5 Interfacing CHORUS/MiX V.4 Servers **16**

5.1 Experimental OO platform 17

5.2 Validation of the COOL Model 17

5.3 Performance Experiments 17

6 Conclusions **18**

1 Introduction **21**

2 COOL–IDL Language **21**

2.1 Interface Definition 22

2.2 Interface Implementation 22

2.3 Interface/Implementation Binding 23

3 Interface Function Invocation **23**

Preface

This paper is a collection of two papers describing COOL, the CHORUS CORBA compliant framework, and COOL–IDL, the interface definition language used for programming distributed applications on COOL.

The first paper is entitled: "COOL: The CHORUS CORBA Compliant Framework". It was presented at the COMPCON Conference, San Francisco, March 1984. It describes the COOL family of run-time support environments based on the OMG-ORB architecture.

The second paper is a position paper entitled "COOL–IDL Extensions to C++ to Support Distributed Programming". It was presented at the Advanced Workshop on Distributed Computing of the USENIX C++ Conference, Cambridge, April 1994. It describes the COOL extensions to C++ to support interface programming on COOL.

COOL: The CHORUS CORBA Compliant Framework[1]

C. Jacquemot, F. Herrmann, P. S. Jensen — *Chorus systèmes*
P. Gautron, J. Mukerji — *Novell/USG*
H.G. Baumgarten, H. Hartlage — *Siemens AG*

Abstract

COOL is a CORBA compliant object-oriented framework for programming services on the CHORUS microkernel. It provides a set of basic facilities (e.g. remote invocation, persistence, migration, ...) to support the creation and execution of services ranging from operating system components to high level applications. The creator of a service can specify a set of facilities that are required by the service. This information is used to select an efficient implementation of the runtime system providing only the necessary facilities. An IDL (Interface Definition Language) called COOL–IDL is used to specify interfaces for the services. COOL–IDL is an extension of C++, consisting of many of the OMG–IDL constructs added to C++. The COOL–IDL compiler generates C++ code for stubs and dispatchers. These are used together with the corresponding runtime system to execute the service as efficiently as possible.

1 Introduction

The OMG (Object Management Group) CORBA (Common Object Request Broker Architecture) is an object-oriented architecture for supporting multi-vendor applications in heterogeneous distributed environments. Support for heterogeneity was the primary motivator of the CORBA specification [5]. CORBA provides the foundation for combining the client/server and object-oriented paradigms. These two paradigms used together have proved to be an excellent approach for building robust distributed applications.

The concept of application has evolved to cover an ever larger class of programs. In distributed operating systems based on microkernel architectures, an operating system is a system application consisting of a set of system servers. On the one hand different system personalities can cooperate on top of the microkernel. On the other hand servers of a same system can be spread out across multiple sites. The CHORUS implementation of UNIX SVR4, CHORUS/MiX V.4 (Modular UNIX), is a typical example of this latter approach: several servers (actors in CHORUS terminology) cooperate to provide the UNIX personality.

Interfaces based on the client/server and object-oriented paradigms can be used in several places: between the microkernel and the operating system(s), between different system personalities on top of the microkernel, between servers of a same operating system, and between users's applications and

[1]This work is partially supported by the European Community under ESPRIT Project 6603 "OUVERTURE".

the underlying system(s). A same formal architecture can be used for providing the interfaces at these different places. But it is necessary to adapt the instance of the architecture to specific requirements of the place where it is used. For example, efficiency, scability and small memory footprint are important in distributed operating systems while CORBA-based user level applications are willing to trade off some efficiency and size for added functionality like heterogeneity.

The COOL (CHORUS Object-Oriented Layer) [4] framework aims at providing a range of object management services ranging from simple and efficient ones for operating system builders to sophisticated environment for application developers. The COOL runtime support environments are based on CORBA and are implemented directly on the CHORUS microkernel. Different versions of the runtime exist to support different levels of functionality. Currently three versions can be generated according to the application requirements and a fourth version has been designed. Specifications are written in an IDL (Interface Definition Language) called COOL–IDL. The same object model is used for all versions of runtime in the family.

We used the CHORUS/MiX V.4 servers as our prototype testbed. This influenced many of the design choices that were made while building COOL. For example the need for providing a clean usable programming environment caused us to design a framework in which interfaces and implementations cooperate without losing their separate identities. This also drove the decision to seamlessly integrate the COOL–IDL with C++, our implementation language of choice.

The rest of the paper is structured as follows. The COOL–IDL language and the generic COOL framework are first presented. Then the COOL framework components and their application on CHORUS/MiX V.4 are described. Finally we share some of our experiences and results of experiments on the prototype version of CHORUS/MiX V.4 system built using this framework.

2 COOL–IDL

COOL–IDL is an IDL based upon C++ as the programming vehicle and OMG–IDL as the formal interface description language. COOL–IDL is a strict extension of C++ and the COOL–IDL precompiler generates standard C++ code corresponding to the COOL–IDL definitions. The mapping of COOL–IDL to C++ is not the same as the one that OMG is likely to adopt as a standard, although the OMG standard binding could be used in one of the instances of environments built using the COOL framework.

In this section we describe the language and the programming model that form the basis for COOL–IDL. Then we present the syntax and semantics of COOL–IDL. Similarities and differences between the COOL–IDL and the OMG models, and between the COOL–IDL extensions to C++ and the OMG–IDL mapping to C++ [2] are also presented.

2.1 COOL–IDL Language

The OMG document specifies a formal object model similar to the C++ object model. These two models are not fully compatible. Our approach is slightly different. The COOL–IDL specification is based on our belief that the ability to reuse existing C and C++ code will be a primary determinant in the acceptance of an interface language. And we conjecture that a first step towards the acceptance of COOL–IDL will be through the use of interfaces encapsulating existing message based communication protocols in exisiting applications that are constructed using a client/server paradigm.

Most aspects of the OMG–IDL are included in COOL–IDL as pure extensions to C++, or are adapted to fit C++ syntax and semantics. Some of the more esoteric aspects of the OMG–IDL were deliberately not included in COOL–IDL. We made pragmatic choices about what aspects of the OMG–IDL specifications are incorporated in COOL–IDL driven mostly by experience gained from experimenting with the prototype CHORUS/MiX V.4 system.

2.2 COOL–IDL Programming Model

The programming model supported by the COOL–IDL is the client/server paradigm. A server consists of an arbitrary number of objects which implement arbitrary services. These objects can be invoked concurrently or sequentially by different applications. A client is an object residing in an application that performs a (remote) invocation on an object in the server.

The implementation of a server is conceptually distinct from its invocation by a client. The server encapsulates the implementation of services it provides and export a well-defined interface to its clients. Clients invoke the server through this interface and do not have any knowledge of the internal structure and implementation of the server.

The C++ class mechanism supports the client/server model to a certain extent. A generic interface can be implemented as an abstract base class, while derived classes, together with the late binding mechanism, provide different implementations. In this model, in order to access a server's functionality, a client needs to know only about the base class of the server.

The COOL–IDL programming model extends the C++ programming model to support distributed computation:

- the implementation of a server is separated from its invocation by a client
- the implementation inheritance tree is decoupled from the interface inheritance tree
- different policies can be implemented for the same interface by different servers
- the client interface object and the corresponding server implementation object are bound at run-time
- different interfaces can be implemented (and exported) by the same server

The COOL–IDL programming model also extends the usual message based model for distributed computation:

- a client invocation of a server is a typed procedure-oriented invocation instead of untyped message passing
- communication protocols can be optimized based on the location of the servers

In our model, to gain access to a server, a client must first acquire an *interface object* associated with the server. An interface object is an instance of a *static* interface definition, static in the sense that the definition is fully specified at compile-time and cannot be modified dynamically at run-time. This model is a restriction of the OMG model that supports dynamic creation and invocation of interfaces.

There is an Object Interface Descriptor Repository (OIDR) from which one can obtain the necessary interface objects. The role of the OIDR is to handle *descriptors* for interface objects. In the current specification, these descriptors are kept as simple as possible. They are tuples of global references and type names. Another way of acquiring interface objects is to have them passed as parameters of interface function invocation. Identical semantics are guaranteed for these objects, irrespective of whether they are obtained from the OIDR or passed as arguments by some other server.

The OIDR is different from the OMG Interface Repository. They are similar in that both store structural patterns. But the Interface Repository stores information relating to *interface definitions* while the OIDR stores information relating to *interface objects*. This information is stored in the OIDR at run-time (typically by a server by exporting its interfaces to the OIDR).

2.3 COOL–IDL Syntax and Semantics

Interfaces are defined using the COOL–IDL. An COOL–IDL interface definition is similar to a C++ class definition in which all its members are public. The difference between an interface and a class relates to how they are implemented. Each class definition has an implementation associated with it. In the case of an interface, although the client and the server share the interface definition, the implementation of the associated implementation class is located only in the server. In the current COOL–IDL specification this sharing is static and is typically done at compile time through shared header files.

An interface object is an instance of an interface definition. A server binds an interface object to a class object implementing that interface (`binding` operation). The server may store the binding information in the OIDR (`export` operation) or fill an interface invocation argument (`out` argument of the interface function invocation). A client that wishes to access the server initializes an interface object by invoking the OIDR (`import` operation) or through an `out` argument of an interface invocation. The client can then use the interface object to access the server.

A COOL–IDL interface is specified by the keyword `interface`, followed by the name of the interface and a set of definitions. For example, the declarations:

```
interface OpenFile {
  int read (in Buffer, in JobControl, out long);
  int write (in Buffer, in JobControl, out long);
};

interface NamedFile {
  int open (in PathName, in int, out OpenFile);
};
```

specifie that:

- `OpenFile` is an interface with two member functions, `read` and `write`, which have three parameters, respectively of type `Buffer`, `JobControl` and `long`, passed from client to server (`in`) or from server to client (`out`)
- `NamedFile` is an interface with a single member function which have three parameters, one of them being an interface object of type `OpenFile` passed from server to client.

There is no distinction in the way C++ objects and interface objects are passed as arguments. Semantics of parameter modifiers (`in`, `out`, `inout`) and of return values are those specified for the proposed OMG–IDL mapping to C++.

A server must bind an implementation to an interface before passing it to the OIDR or to the interface function invocation. An implementation is typically a regular C++ program that is run in a process constituting the server. At runtime an interface object is created and bound to the implementation object in the server before the interface object is exported either to an OIDR or to a client through an interface function out parameter. The operations used to set up the communication between a client and a server are described below. This communication is managed by the underlying object-oriented runtime system.

A COOL–IDL interface is implemented by a C++ class. The access specifier `interface` has been added to the C++ grammar to specify the interface(s) that are implemented by a C++ class. The following class declaration shows an example of a class `Impl_OpenFile` that implements the interface `OpenFile` above:

```
// -- Implementation of OpenFile
class Impl_OpenFile {
  // ...
public:
  int read (const Buffer&, const JobControl&, long&);
  int write (const Buffer&, const JobControl&, long&);

interface:     // access specifier
  OpenFile;    // interface to be implemented
};
```

Public member function declarations with signatures that match those of the interface functions must be found in the class scope (class, base(s) class(es) but not global scope) of the class implementing the interface.[2] Two signatures are said to match if the names of the two functions are the same, the number of parameters, their types, modifiers and order of appearance in the parameter list are the same, and the return value is of the same type.

When parsing an interface specification, the COOL–IDL pre-compiler checks that all the interface functions are defined by the implementation class[3] (i.e. have a corresponding implementation function with a matching signature). In COOL–IDL, interface and implementation obey a conformity rule, as opposed to the inheritance relationship as proposed in [2].

When parsing an `interface` access specifier, the COOL–IDL pre-compiler generates a conversion operator for the corresponding interface. The data layout of the implementation class is not modified.

A server binds an implementation object to an interface object by assigning a class instance to an interface instance. For example:

```
OpenFile      openfile;      // interface instance
Impl_OpenFile impl_openfile; // implementation instance
```

[2]More elaborate interface function renaming in an implementation class is supported by COOL–IDL but is not described in this paper.

[3]The list of member functions declared in an implementation class is not limited to the functions of a specific interface. In particular, different interfaces with different function names can be implemented by a same class, and the class may also define member function that do not match any of the interface member functions.

```
openfile = impl_openfile;    // binding operation
```

In accordance with the standard C++ rules for type conversion, the assignment above causes the conversion operator for the interface to be called. This operator initializes the interface object with information related to the server object.

After the binding operation is processed, the interface object can be passed by the server to the OIDR or as argument to an interface invocation.

3 Generic COOL Architecture

For each interface definition, the COOL–IDL pre-compiler generates a C++ class with the same name. Clients operate directly on interface objects: an interface function call is a call on a function of the interface object. The interface object corresponds to the *stub* part in CORBA. An interface object is an instance of a stub class.

For each implementation class (distinguished from a standard C++ class by its `interface` access specifier), the COOL–IDL pre-compiler generates a C++ skeleton class, which corresponds to the *skeleton* part of the CORBA, and the conversion operator used to bind the implementation object to an interface object. The code generated for the skeleton class is statically linked with the server. The skeleton class is the server counter-part to the client stub class. It reifies the stub requests in a form that can be processed by the implementation functions.

The stub and skeleton classes are completed by a mechanism to turn message based invocations into local invocations and vice-versa. This mechanism is used to optimize communication when the server and the client are co-located in the same address space.

A runtime library implements the communication between the stub, skeleton and server parts. Several C++ classes compose this library. They are stub and skeleton classes, invocation classes, memory management classes and cluster class. The following considerations influenced the design of the library:

- The code generated for the client is kept strictly independent of information relating to the server.
- Each functionality (notably invocation and memory management) can be re-implemented without requiring the other components to be modified.
- The library components is designed to be efficient and to reduce complexity of the pre-compiler generated code.
- Making it possible to provide an efficient, flexible and transparent mechanism for co-location optimization [2].

3.1 Stub and Skeleton Classes

Consistent with the CORBA model, the C++ classes generated from interface definitions all share a common base class, named `Object` in OMG–IDL and `Interface` in COOL–IDL.[4]

Abstract client and server classes derive from the `Interface` class. The stub class generated by the COOL–IDL pre-compiler derives from the abstract client class. The skeleton class derives from the abstract server class. The class hierarchy for the example shown §2.3 is as follows:

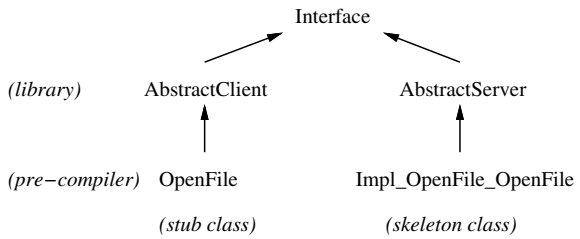

Figure 1: Class Hierarchy

All data related to interface management are encapsulated in the base class and most of the code generated by the COOL–IDL pre-compiler consists of functions calling base class functions.

Interface functions are turned into virtual member functions of the stub class. By default, these functions perform remote procedure calls to the corresponding skeleton class. Defining these functions as virtuals allow efficient and clean co-location optimization. This mechanism is described in §3.2.

For each pair (implementation class, interface) parsed, the COOL–IDL pre-compiler generates a skeleton class. A skeleton class is specific in the sense that it is never instantiated (see §3.2). There are two distinct parts in the class definition:

- a static `dispatch` function, reifying the stub requests before calling the appropriate implementation functions.
- virtual declarations of functions similar to the interface function declarations, delegating the calls to the implementation (stub functions, skeleton functions and implementation functions all share the same signatures). The skeleton class virtual table is only used for co-location optimization.

3.2 Optimizing Co-location Invocations

An obvious optimization for co-located client and server is to use direct procedure calls instead of RPCs to invoke server functions. This optimization was a particularly important consideration in the design of the runtime system.

To achieve this, we rely on localization mechanisms supplied by the microkernel (§3.3) and on the C++ virtual function mechanism to convert from remote calls to direct calls and vice-versa. This latter mechanism is shown below.

[4] All the generated global names that could conflict with user's definitions has been prefixed with `_I_`. Class names are not prefixed in the paper. C++ name spaces, when available, will provide the appropriate mechanism to solve this issue.

The default is to perform remote procedure calls. The virtual member functions of the stub class encapsulates IPC calls to the skeleton dispatch mechanism (figure 2a).

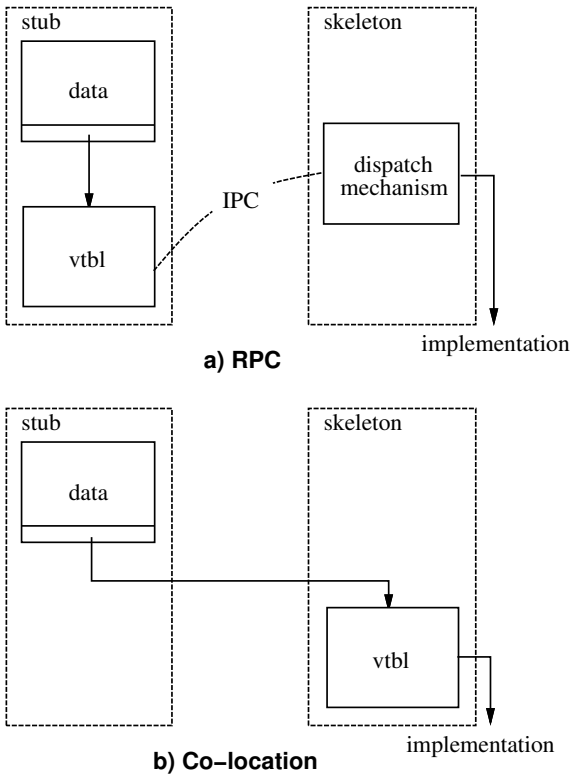

Figure 2: Interface Invocation Mechanism

Server location is implicitly checked when an interface object is copied, when an interface object is passed as `out` argument of an interface call and at each interface invocation in systems in which migration is supported.

When the client and the server are located in the same address space, the private virtual pointer of the interface object is modified to directly point to the virtual table of the skeleton class[5] (figure 2b).

Reassigning the virtual pointer of the interface object is possible because the COOL–IDL pre-compiler guarantees that the memory layouts of the stub and skeleton classes are strictly identical. Skeleton virtual member functions operate directly on interface objects data. There is no necessity to instantiate the skeleton classes.

3.3 Cluster Class

Supporting migration and persistence requires the definition of the granularity of the entities that are to be migrated and stored. In general COOL objects are fine-grained objects. Previous experiments with COOL applications [4] show that inter-object dependency results in the necessity to group objects that are closely related and hence natural to migrate and store together. The notion of *cluster* captures

[5]The pre-compiler generates code using multiple inheritance and so multiple virtual pointers and tables. In our implementation, pointer updating is automatically managed by the C++ compiler.

this concept of group of closely related objects in the COOL model. A cluster constitutes a unit of migration and persistence. Direct references between related objects from within the same cluster are allowed.

Introduction of the concept of cluster impacts the naming mechanism. In particular, an implementation object must be formally identified by a global reference unique to its universe. The COOL model defines this global reference as composed of the global reference of a cluster and a unique identifier inside the cluster. The unique global reference is part of the data of an interface object (it is declared in the `Interface` class). It is initialized by the server when an implementation object is bound to the interface object (§2.3). The COOL runtime library relies on CHORUS communication facilities (port, group, capabilities) to implement cluster references.

3.4 Invocation Classes

Invocation classes are used to encapsulate the operations involved in invoking a function of an interface object. The role of the invocation classes is threefold:

- to allocate the request and reply communication buffers (§3.5)
- to format the message layout for the communication protocole
- to perform IPC calls between the client and the server

There are three classes that serve this purpose, `ClientInvoc` and `ServerInvoc` deriving from `Invoc`. Splitting the functionality into three classes clarifies the implementation:

- CHORUS message management is shared by the client and the server classes
- parameter marshalling is shared by the client and the server classes
- the `invoke` function is a part of only the client class
- the `reply` function is a part of only the server class

3.5 Memory Management Classes

Memory management is a critical part of the RPC mechanism. The application itself can supply its own mechanism that can be used for the implementation of the communication framework. This is the case, for example, for interfaces defined between CHORUS/MiX V.4 servers, where the standard UNIX kernel memory allocators are used.

4 The Family of COOL Supports

The previous section described the generic COOL framework. We have built several applications using this framework including a prototype version of CHORUS/MiX V.4. In doing so we have discovered the following to be important architectural considerations:

- Objects representing interfaces must be fine-grained: Interface objects are intented to replace pointers in a monolithic system when remote access is required and to be the core naming mechanism. Size of these objects influence the overall system size. Only the information required for a specific configuration should be stored in these objects.
- Co-location optimization: Interface invocation can be optimized and argument copying can be avoided when the client and the server share the same address space. Co-location can be a static configuration parameter of the application or can be detected dynamically.
- Support for migration and persistence: Client and/or server migration requires to dynamically update the interface invocation mechanism according to the new locations.
- Support for a full object-oriented layer interfacing the microkernel: The COOL framework is intended to be the core foundation for building future subsystems on top of the CHORUS microkernel.

We identified the naming and localization mechanisms as the critical parts of the architecture. Then we came to the conclusion that the most efficient way to support these applications was to create different systems within the same framework. The need for four instantiations is based on specific properties, and they are as follow:

- COOL–L: support for `Local` objects created in the same address space at fixed addresses. This is useful for example in building a single-site implementation of CHORUS/MiX V.4 where all the servers are always co-located.
- COOL–LD: support for objects in `Distributed` address spaces at fixed addresses. This is used for example in a multi-site implementation of CHORUS/MiX V.4 where objects are always located on the site where they are created.
- COOL–LDM: support for `Migratable` objects mapped in `Distributed` address spaces. This is useful for example in a distributed implementation of CHORUS/MiX V.4 where objects (processes, pipes, SV messages queues, SV semaphores, ...) can migrate.
- COOL–LDMP: support for `Migratable` and `Persistent` objects mapped in `Distributed` address spaces. This is useful for building groupware applications on top of COOL or an implementation of CHORUS/MiX with checkpointing facilities.

Figure 3 pictures the family of supports based on the COOL framework.

Implementation of the naming and localization mechanisms is encapsulated in the library class `Interface` (§3.1) but also affects the pre-compiler generation of the stub and skeleton classes (§3.1).

Only three members of the family are currently implemented. The COOL–LDM member is intented to be the runtime support for a more elaborate implementation of CHORUS/MiX V.4, with more stringent performance requirements than for COOL–LDMP. At present we lack sufficient experience with COOL–LDM to report on it. Consequently this section describes only three members of the family.

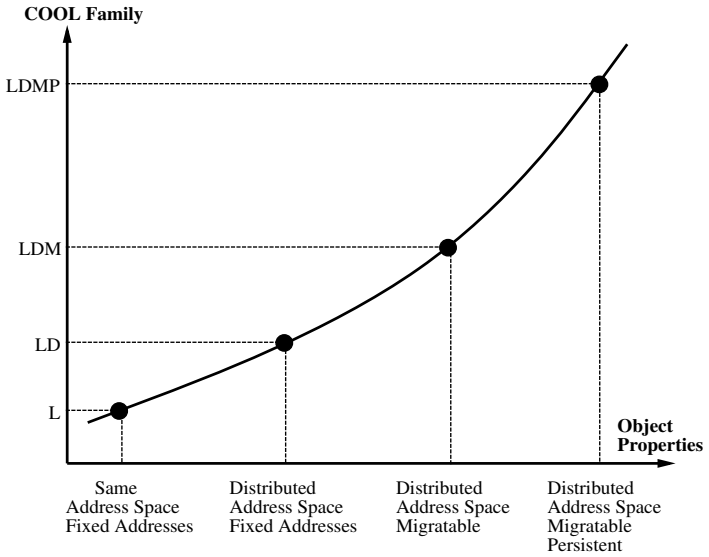

Figure 3: COOL family

4.1 COOL–L Architecture

COOL–L provides the minimal runtime support of the COOL programming model when objects are created at fixed addresses within the same address space. Using interfaces on top of COOL–L enables substitution of procedure calls for message passing when single-site CHORUS system actors are created (they share the same system address space) and will never migrate.

All objects belong to the same cluster (§3.3). Inside a cluster, an object's identity is completly determined by its address and a global reference to it can be reduced to its address in the address space.

An interface object is a tuple of one integer and two pointers:

- The integer typename is used for internal type verification. This field could be eliminated if no type verification is required.
- The address of the implementation object.
- The address of the function used for processing co-location optimization. This address depends on the location of the implementation object class, and is hence bound at run-time.

Co-location optimization is performed when an object implementation is bound to an interface object (§3.2). Stub functions are necessary to compile the client as an independant module but are never called. Their implementation is reduced to a call to an error handler.

The dispatch function of the skeleton is not generated since no RPC is performed.

Skeleton functions are C++ virtual functions that are invoked on interface objects. The address of the implementation object is part of the data contained in an interface object. A call to a skeleton function forwards this call to the corresponding function in the implementation object. Argument passing can be further optimized at compile time by specifying an option to the precompiler to use reference instead of bitwise copying wherever possible. Bitwise copy guarantees that local and remote

behaviors are identical when local and remote semantics are required to be identical. It is up to the programmer to guarantee similar semantics when argument passing by reference is opted.

4.2 COOL–LD Architecture

COOL–LD provides the minimal runtime support of the COOL programming model when objects are created at fixed addresses in distributed address spaces.

Using COOL–LD interfaces enables substitution of procedure call in place of message passing when objects are co-located, while using appropriate RPC mechanisms underneath when objects are not co-located.

COOL–LD has been used to implement inter-server communication in a prototype version of CHORUS/MiX V.4. Object invocation performance has a very large impact on the global performance of the system. Hence emphasis was placed on argument passing, memory management, library classes specific to CHORUS/MiX V.4 functionalities and performance optimizations.

A system actor defines an address space and a single cluster is associated with each actor. Inside a cluster, an object's identity is completly determined by its address. A CHORUS port is associated with each cluster. Global references are pairs of the cluster's port unique identifier and the object's address within the cluster.

When a message is received on the cluster's port, the function attached to message management localizes the implementation object and calls the skeleton dispatch function associated with the interface object.

An interface object is a 4-tuple consisting of one integer, one global reference and two pointers:

- The typename, used for internal type verification.
- The global reference of the object implementation.
- The address of the dispatch function in the skeleton class. This address is always valid since the server code is statically loaded.
- The address of the function used for processing co-location optimization.

Stub functions perform RPCs to the implementation objects. Co-location is checked when an interface object is initialized (from the OIDR or as argument of an interface call) with the help of the microkernel.

Both the dispatch function and the virtual functions are generated for each skeleton class. Use of these functions is described in §3.1.

As mentionned earlier, argument passing was studied during our experimentation on CHORUS/MiX V.4. By default, arguments are pushed into or popped from the communication frames. This extra copy could become too expensive. An alternative is to implement library classes in order to pass data descriptors instead of raw data in the communication frame. Data are then explicitly copied when needed using distributed copy functions provided by the microkernel. This mechanism is used for the CHORUS/MiX V.4 implementation of the `read` and `write` system calls (see §5) but also, for example, to internally manage pathnames.

4.3 COOL–LDMP Architecture

COOL–LDMP (also referred to as COOL v2 [4]) provides a distributed framework for the COOL programming model with persistent and migratable objets. It is designed to support sophisticated object-oriented applications. COOL–LDMP was in fact the first member of the COOL family and influenced the overall design of the framework.

A cluster is the unit of persistency and migration. When mapped into an address space, a cluster is a collection of virtual memory regions backed into segments (files) by means of external mappers. The mapping may be to an arbitrary address, into an arbitray address space created at an arbitrary site. When an unmapped object is invoked through an interface object, its cluster is mapped into an address space and then the invocation is executed.

The COOL–LDMP naming mechanism provides a means to localize an object that is mapped into an address space or swapped-out on secondary storage. A CHORUS communication port is associated to each address space and a CHORUS communication port group is associated to each cluster. When a cluster is mapped into an address space, the corresponding port is inserted in the cluster's group. It is removed from the group when the cluster is unmapped. Global references are doublets of the cluster's group unique identifier and a global value incremented at each object creation inside the cluster.

A generic `runtime-to-language` layout adapts the implementation language to the runtime support. An upcall mechanism allows the runtime support to access information only known at language level on a per classe basis. It is used for a variety of purposes, including persistency (pointer handling) and invocation.

An interface object is a doublet of one integer and one global reference:

- The typename, used for internal type verification
- The global reference of the object implementation

Moreover, unique names are built for cluster's segments from the cluster identifier. Localizing an unmapped cluster is no more than localizing a file in the storage's site of the cluster.

Stub functions perform RPCs to the implementation object. Objects are migratable and require to check co-location of the implementation object at each invocation.

Both the dispatch function and the virtual functions are generated for each skeleton class. During its invocation, the implementation object is locked to avoid its migration.

5 Interfacing CHORUS/MiX V.4 Servers

CHORUS/MiX V.4 system functionality is spread among several components: the CHORUS microkernel and system servers (process manager, file manager, stream manager, ...). All servers are run as individual actors in supervisor mode and share the same system address space. Communication between servers is based on CHORUS IPC.

UNIX applications invoke CHORUS/MiX V.4 services through system calls. These system calls trap into the process manager. They are directly served by the process manager (as `signal` handling) or redirected to another server via RPC (object manager for `open` or `creat`).

A prototype of the CHORUS/MiX V.4 subsystem with COOL–IDL interfaces between servers has been implemented and tested using the COOL–L and COOL–LD supports. We wanted to evaluate the usefulness of the COOL approach for developing operating systems using these prototypes. For this study, a set of system calls (from the top 10 system calls of normal workload) were selected as testbed for conversion to interface invocations.

The prototype was built with the following goals:

- Providing an experimental Object-Oriented (OO) platform.
- Validating the COOL model.
- Running performance measurements.

5.1 Experimental OO platform

Both COOL–L and COOL–LD were used as runtime support for CHORUS/MiX V.4 implementation. Different specific developments were required, including file organization (interfaces definition, stub and skeleton implementations), compiler tools and libraries. CHORUS/MiX V.4 servers are written either in C++ or in C. Implementing object servers in C++ allows easy integration of interfaces with UNIX code. Finally, the COOL model allows incremental conversion to interfaces while preserving the capability to fall back on legacy message passing. The converted interfaces can be switched back and forth between using COOL interfaces and legacy message passing at runtime, thus facilitating comparison of performance measurement on an interface by interface basis.

5.2 Validation of the COOL Model

Native implementation of CHORUS/MiX V.4 servers was based on semantics of separate actors. The COOL model does not interfere with the actor model. However in case of co-location optimization, the client directly calls skeleton functions (§3.1): the client thread runs server's code within the client context. Obeying the same semantics requires specific system calls to migrate threads between the client and server actors. This code is added to the skeleton functions, as prologue and epilogue to the calls to the implementation. These system calls impact system performance and can be disabled using compiler directives if servers are carefully written to take this constraint into consideration.

5.3 Performance Experiments

Interfaces for the `read/write` (R/W) system calls were especially implemented because they are in the top 10 system calls and because the same interface is implemented by two servers (in file and stream managers) and used indistinguishably by the same client (in process manager).

Performances were measured on pipe R/Ws using COOL–L. The figure 4 below pictures pairs of R/Ws in terms of transferred data size and shows the gain, in terms of percentage, resulting from using interfaces compared to standard CHORUS/MiX V.4. For example, this figure shows that one R/W of 1 Kbyte is 17% faster.

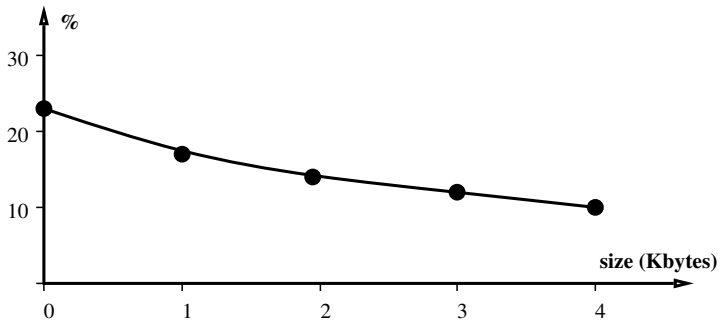

Figure 4: Gain on R/Ws

6 Conclusions

The COOL framework has been successfully used for both high level application programming and for operating system services programming. At the operating system end of the spectrum COOL–L and COOL–LD have been used successfully in the CHORUS/MiX V.4 system. We are exploring the use of COOL–L within the CHORUS microkernel itself. In the prototype systems put together so far we have been successful in limiting the cost in performance and memory commensurate with the level of functionality.

At the other end of the spectrum COOL–LDMP has been used in groupware applications, as the CIDRE application developed by the SEPT, a research laboratory of France Telecom. CIDRE is designed as a distributed platform for the management of intelligent documents [4]. The COOL–LDMP applications were all written from scratch on the COOL framework. The first test of incremental use of COOL–LDMP will take place in the near future when we use COOL–LDM, a lighter weight version of COOL–LDMP, which supports migration but not persistence for programming some of the CHORUS/MiX V.4 servers.

The COOL framework is an excellent vehicle for designing and testing server architectures through rapid incremental prototyping of interfaces. The mechanism for constructing more complex interfaces by aggregating simple interfaces facilitates this. The same mechanism makes it possible to incrementally introduce the use of COOL interfaces in an existing system. We have succesfully demonstrated this by introducing the use of interfaces into the legacy UNIX code in the CHORUS/MiX V.4 system. This has provided us the means to port the system over a period of time in an evolutionary fashion, to the COOL framework.

In the existing version of the framework different versions of the same interface (i.e. COOL–LD, COOL–L, COOL–LDMP, ...) do not interwork cleanly with each other. We are exploring the most cost effective way in terms of performance and size to remedy this situation. The principle of not incurring a cost for a functionality that is not needed will be adhered to in this evolutionary work.

On the whole the experience with the COOL framework suggests that it is a qualified success. It has achieved or come close to achieving all that we set out to do at the inception of the work. However, in course of its development we have discovered new opportunities for improvement, like the opportunity to get the different architectures that fit within the framework to interwork with each other. Work on such opportunistic embellishments remains to be done. Meanwhile, we are actively introducing the versions that have been successfully prototyped into everyday use as and when the opportunity presents itself.

References

[1] R. Lea, C. Jacquemot, E. Pillevesse, "COOL: System Support for Distributed Programming", *Communications of the ACM*, Vol. 36, N. 9, September 1993.

[2] M.A. Goulde, "Tomorrow's Microkernel-based Unix Operating Systems", *Open Infomation Systems*, Vol. 8, N. 8, August 1993.

[3] HyperDesk Corporation, "Revised Submission in Response to the OMG RFP for a C++ Language Mapping", *OMG TC Document 93-9-2*, September 1993.

[4] OMG, "The Common Object Request Broker: Architecture and Specification", *OMG Document Number 91.12.1*, 1991.

[5] Chorus Team, "Overview of the Chorus Distributed Operating System", *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle (USA), April 1992.

[6] G. Hamilton, M. L. Powell, J. G. Mitchell, "Subcontract: A Flexible Base for Distributed Programming", *SMLI TR-93-13*, Sun Microsystems Labs Inc, April 1993.

COOL–IDL Extensions to C++ to Support Distributed Programming [1]

Position paper

C. Jacquemot, F. Herrmann, P. S. Jensen — *Chorus systèmes*
P. Gautron, J. Mukerji — *Novell/USG*
H.G. Baumgarten, H. Hartlage — *Siemens AG*

1 Introduction

The CORBA (Common Object Request Broker Architecture) is an an object-oriented architecture specified by the OMG (Object Management Group) for supporting multi-vendor applications in heterogeneous distributed environments. The CORBA document [5] specifies both an architecture which is based on the client/server model, and an IDL (Interface Definition Language). The IDL is used to specify interfaces that are implemented by a server and invoked by clients. The OMG–IDL is a specification language based on the object-oriented paradigm. Its constructs are mapped to an implementation language such as C or C++in which the distributed application is implemented. Interfaces specified in OMG–IDL are supposed to behave similarly irrespective of the implementation language that is used to implement the application.

Specifying the interface and the implementation separately in different languages is sometimes inconvenient because types defined in the implementation language cannot be used in the interface definitions without redefining them in the IDL. Since our project is implemented entirely in C++ we decided to explore the possibility of incorporating the most important OMG–IDL constructs as extensions to C++. The result of this effort is the IDL called COOL–IDL that we describe in this paper. Experience in using COOL–IDL interfaces for programming server interactions in Chorus/MiX V.4, which is a microkernel based multi-server implementation of UNIX SVR4, validated this approach.

COOL–IDL forms an integral part of the COOL (CHORUS Object Oriented Layer) framework [3, 4], and hence the code generated from COOL–IDL specifications uses Chorus IPC [1] based facilities provided by the COOL generic runtime for remote object invocation. However, it is possible to use an alternate IPC mechanism like sockets by modifying the COOL generic runtime library if one so desires.

2 COOL–IDL Language

The design of COOL–IDL is based on the belief that the ability to reuse existing C and C++ code will be a primary determinant in the acceptance of an interface language. COOL–IDL is an IDL based

[1] This work is partially supported by the European Community under ESPRIT Project 6603 "OUVERTURE".

upon C++ as the programming vehicle and OMG–IDL as the formal interface description language. COOL–IDL is a strict extension of C++ and the COOL–IDL pre-compiler generates standard C++ code corresponding to the COOL–IDL definitions.

Interface specifications are written in COOL–IDL and placed in .idl files. Regular C++ header files can be included by .idl files thus making it possible to use implementation objects as parameters in the interface specifications. When a .idl file is compiled corresponding C++ header information and C++ code is generated only for the contents of the .idl file (and not for any files included by it). These files can then be compiled together with other C++ or C implementation files to build the server and client load modules.

Most aspects of the OMG–IDL are included in COOL–IDL as pure extensions to C++, or are adapted to fit C++ syntax and semantics. Some of the more esoteric aspects of the OMG–IDL are deliberately not included in COOL–IDL. We made pragmatic choices about what aspects of the OMG–IDL specifications are incorporated in COOL–IDL driven mostly by experience gained from experimenting with the prototype CHORUS/MiX V.4 system.

2.1 Interface Definition

In COOL–IDL an interface is specified by the keyword `interface`, followed by the name of the interface and a set of definitions. For example, the declarations:

```
interface OpenFile {
  int read (in Buffer, in JobControl, out long);
  int write (in Buffer, in JobControl, out long);
};

interface NamedFile {
  int open (in PathName, in int, out OpenFile);
};
```

specify that:

- `OpenFile` is an interface with two member functions, `read` and `write`, both of which have three parameters, of type `Buffer`, `JobControl` and `long`, passed from client to server (`in`) and from server to client (`out`) respectively.
- `NamedFile` is an interface with a single member function which has three parameters, one of which is an interface object of type `OpenFile` passed from server to client (`out`).

There is no distinction in the way C++ objects and interface objects are passed as arguments.

For each interface definition, the COOL–IDL pre-compiler generates a C++ class with the same name. An interface object is an instance of this class. §3 describes how clients and servers operate on interface objects.

2.2 Interface Implementation

A COOL–IDL interface is implemented by a C++ class. The access specifier `interface` has been added to the C++ grammar to specify the interface(s) that are implemented by a C++ class. The following class declaration shows the class `Impl_OpenFile` that implements the interface `OpenFile` above:

```
// -- Implementation of OpenFile
class Impl_OpenFile {
  // ...
public:
  int read (const Buffer&, const JobControl&, long&);
  int write (const Buffer&, const JobControl&, long&);

interface:     // access specifier
  OpenFile;    // interface to be implemented
};
```

Public member function declarations with signatures that match those of the interface functions must be found in the class scope (class, base(s) class(es) but not global scope) of the class implementing the interface.[2] Two signatures match if the names of the two functions are the same, the number of parameters, their types, modifiers and order of appearance in the parameter list are the same, and the return value is of the same type. Interface and implementation must always obey the conformity rule stated above.

Corresponding to each `interface` access specifier, the COOL–IDL pre-compiler generates a conversion operator for the corresponding interface. The data layout of the implementation class is not modified.

2.3 Interface/Implementation Binding

A server must bind an implementation to an interface before passing it as `out` parameter of an invocation of an interface function.[3] A server binds an implementation object to an interface object by assigning a class instance to an interface instance. For example:

```
OpenFile      openfile;      // interface instance
Impl_OpenFile impl_openfile; // implementation instance

openfile = impl_openfile;    // binding operation
```

In accordance with the standard C++ rules for type conversion, the assignment above causes the conversion operator for the interface to be called. This operator initializes the interface object with information about server object.

3 Interface Function Invocation

For each interface definition, the COOL–IDL pre-compiler generates a C++ class with the same name. An interface object is an instance of this class. An interface function call is a call of a function of the interface object. An interface object corresponds to a *stub* in CORBA. For each implementation class (distinguished from a standard C++ class by its `interface` access specifier), the COOL–IDL pre-compiler generates a C++ skeleton class (`Impl_OpenFile_OpenFile` for the example above), which corresponds to the *skeleton* part in CORBA. Functions of the implementation class are called from the corresponding functions of the skeleton class.

[2] More elaborate interface function renaming in an implementation class is supported by COOL–IDL but is not described in this paper.

[3] Object interface descriptors can also be exported to or imported from a dedicated external server. This is not described in this paper. The same mechanism is used in either case.

Consistent with the proposed OMG–IDL C++ binding[2], all the C++ classes generated from interface definitions share a common base class. This class is called `Object` in the proposed C++ binding of the OMG–IDL and `Interface` in the case of COOL–IDL. Figure 5 shows the class hierarchy for the `OpenFile` interface. Interface functions are turned into virtual member functions of the stub class.

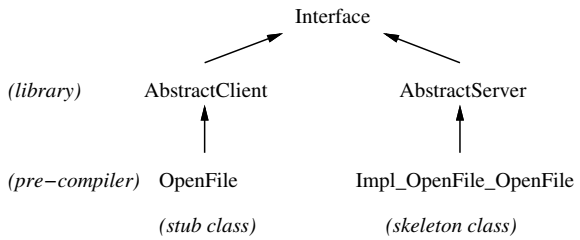

Figure 5: Class Hierarchy

By default, these functions perform remote procedure calls to the corresponding functions of the skeleton class (figure 6a). Interfaces encapsulate the mechanism for transparent invocation of servers

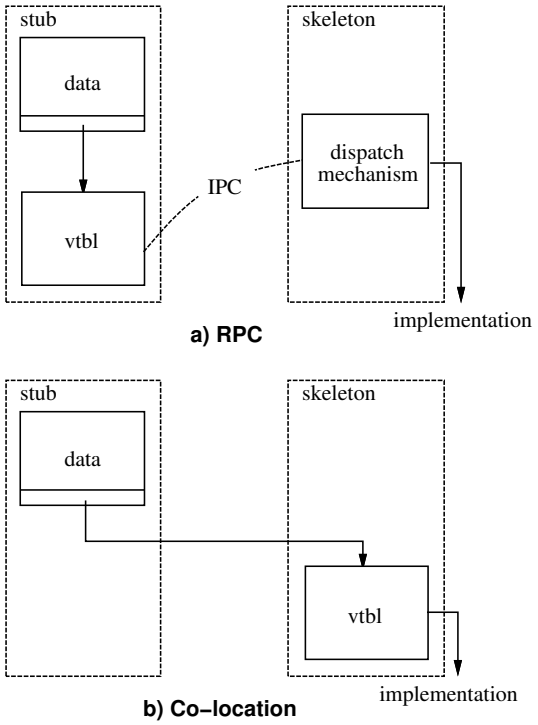

Figure 6: Interface Invocation Mechanism

located anywhere in a COOL distributed system. Experience has shown that for co-located client and server, optimization of the invocation path improves performance significantly. So this optimization was given particularly attention in the design of the COOL runtime system. When the client and the server are located in the same address space, the private virtual pointer of the interface object is modified to directly point to the virtual table of the skeleton class (figure 6b).[4]

[4]The pre-compiler generates code using multiple inheritance and hence multiple virtual pointers and tables. In our implementation, pointer updating is automatically managed by the C++ compiler through the placement syntax of the `new` operator.

References

[1] Chorus Team. "Overview of the Chorus Distributed Operating System", *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle (USA), April 1992.

[2] HyperDesk Corporation. "Revised Submission in Response to the OMG RFP for a C++ Language Mapping", *OMG TC Document 93-9-2*, September 1993.

[3] C. Jacquemot, F. Herrmann, P. S. Jensen, P. Gautron, J. Mukerji, H.G. Baumgarten, H. Hartlage. "COOL: The CHORUS CORBA Compliant Framework", To appear in *Proceedings of COMPCON'94*, San Francisco, March 1994.

[4] R. Lea, C. Jacquemot, E. Pillevesse. "COOL: System Support for Distributed Programming", *Communications of the ACM*, Vol. 36, N. 9, September 1993.

[5] OMG. "The Common Object Request Broker: Architecture and Specification", *OMG Document Number 91.12.1*, 1991.