## CONTENTS

# A distributed object-oriented platform based on DCE and C++

Pier Giorgio Bosco, Giovanni Martini, Corrado Moiso

CSELT
via Reiss Romoli 274 10148 Torino (Italy)
e-mail: {bosco,martini,moiso}@cselt.stet.it

**Abstract**

This paper presents some results on the definition of a distributed processing model based on the ODP reference model and its prototype implementation based on DCE and C++.

Keyword Codes: D.1.3; D.1.5;
Keywords: Object-orientation; Distributed Processing; Distributed Platforms

## 1. INTRODUCTION

The purpose of this paper is to present some preliminary results of a research activity carried on at CSELT on the definition and prototype implementation of a distributed processing environment, based on the ODP prescriptive model [1]. In particular we concentrate here on the computational, engineering and technology viewpoints with major emphasis on the latter two.

The motivation for this research comes directly from the (telecom) operating company needs of mastering the software complexity imposed by the rapidly growing, highly diversified market demands, and consequently, by the necessity of quickly and economically developing and introducing new services.

To achieve such an ambitious, yet strategic to the Network Operators, goal, we are investigating an approach based on a single conceptual model rich enough to express the descriptive, computational and deployment needs of different areas like Intelligent Networks, TMN and Operation Systems: this model tries to unify those concepts of different areas that are sufficiently close one another.

A summary of the current state of the model is given in Sect. 2. The remaining part of the paper is focused on a specific realization of the model, which consists of an API (Sect. 3) and of its implementation (Sect. 4) by means of a widely available software platform (DCE and C++) which we developed as a first step towards a multi-platform implementation.

## 2. THE DISTRIBUTED PROCESSING MODEL

The distributed processing model provides a framework for describing the software structure and behaviour of distributed (telecommunication) systems. The model, based on the object-oriented paradigm according to ODP-RM, provides a uniform description of the structure of different entities (programs, data, network resources,...) involved in a telecom application. The model tackles both the structural and the behavioural/interaction aspects, through the definition of templates and primitives.

**2.1. Structural aspects**

The structure of a distributed system is expressed in terms of objects, interacting through services provided by interfaces. The objects are clustered into software units, called building-blocks (BB), that introduce the following visibility rules on interfaces:

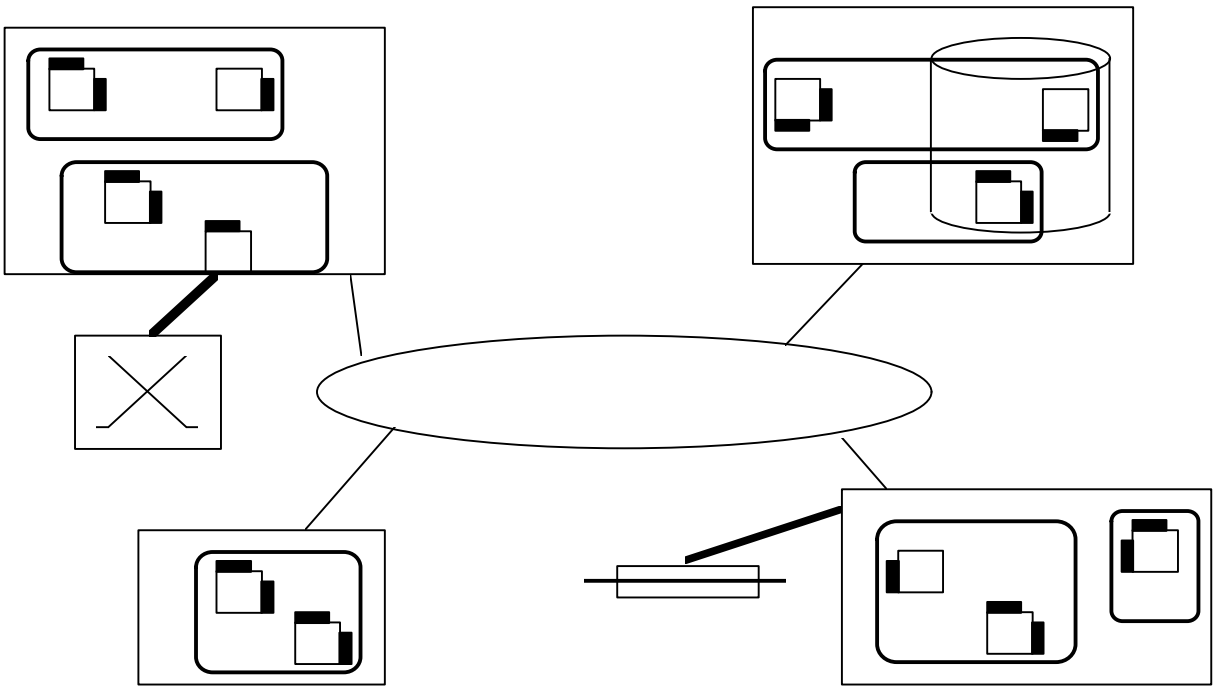
- the interfaces visible outside of a BB are called contracts;
- an object can interact only with interfaces provided by the objects in the same BB or with contracts provided by objects in other BBs.

The motivations for such a structuring, similar to the one adopted in the INA initiative [2], are reported in [3].

A distributed application consists of one or more BBs, each installed on a single host. A BB is a unit of modularity, installation, administration, failure and replication.

A BB (instance) is an instantiation of a (parametric) BB-template: it describes its internal state structure and its behaviour through a set of object templates and specifies the types of the offered contracts.

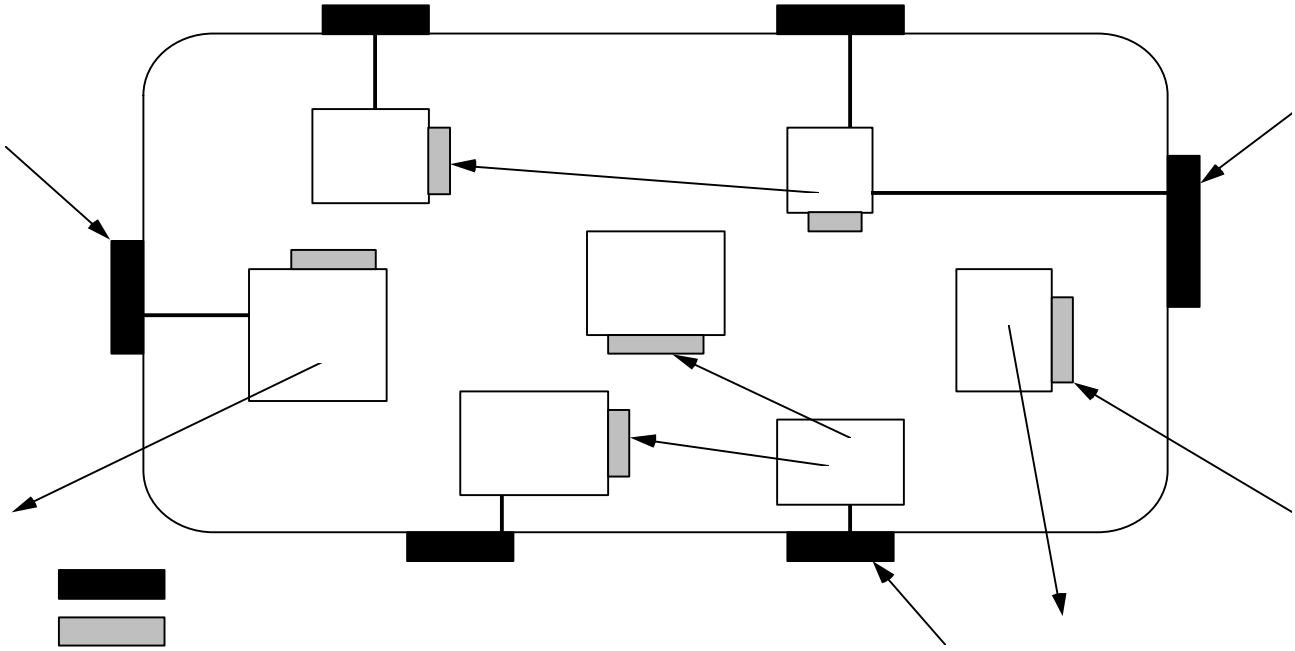
At run-time a BB is a unit of creation/execution of the objects whose templates are included in the corresponding BB template: these objects form a dynamic set. Moreover, a BB is a unit of provisioning of services according to its contracts.



One of the object templates is elected as the template of the manager of the BB, i.e. the object that performs the initialization phase of the BB and provides some management functionalities.

The objects inside a BB instance could be either static or dynamic: a static object has the same lifetime of the BB (it is created when the BB is installed and is never explicitly destroyed); a dynamic one is created by an other object in the BB and may be terminated.

A BB provides a set of contracts, whose types are declared in the template: the contracts, which are (a subset of the) interfaces of objects in the BB, are static (resp. dynamic) if they are provided by static (resp. dynamic) objects. In possible variants of the model, there could be just static contracts.



The structure and the behaviour of the objects in a BB are described by objects templates. The objects are units of functionality and encapsulation that can interact through interfaces. An object template includes the following fields:

- formal parameters, to be supplied when a new instance is created;
- state variable declarations;
- interface declarations;
- methods declarations, including the initialization and the termination phases.

The objects provide a static set of multiple interfaces, one of which is elected as the main object interface. An interface provides to its clients the same kind of services provided by OSI managed objects:

- operations: to invoke methods of the object;
- attributes: to read/write state variables of the object;
- notifications: to require to be notified of something happened in the object.

The main object interface (reference) corresponds to the object identifier and provides some additional services, among which the termination of the object.

The references to interfaces and contracts are denotable values that can be stored in variables and passed as arguments or results when interactions are performed.

### 2.2. Typing aspects

The model has a type system that guarantees the correctness of the interactions among the objects: in a well-typed program there are no requests of operations or notifications to

(dynamically determined) interfaces that do not provide them as well as no accesses to non-visible attributes.

The type system of the model enriches the type system of the languages adopted to describe the behaviour of the objects, with the types which are specific to the model: interfaces, main interfaces, operations, attributes,...

The type system introduces a subtype relation, based on a subtype relation among the interfaces: as in many other distributed object-oriented models, an interface type is a subtype of another one if it provides more operations, attributes and notifications and less "results".

### 2.3. Behavioural aspects

The behavioural aspects considered by the model can be grouped in the following way:

- constructs to manipulate instances of BB and objects (creation, termination,...);
- constructs to perform interactions between objects ;
- constructs to control the concurrent activities;
- constructs to access the architectural services (e.g. the Trader).

*Manipulation of instances*

In a system there can be at the same time several instances of the same BB template, possibly on the same host. The request to create a new BB instance needs (at least):

- the name of the BB template;
- the list of actual parameters, used to configure the instance;
- the name of the target host.

The following actions are performed when a BB instance of template T is created on host H:

- production of the executable code (according to some makefile-like commands in the BB template) suitable to H;
- loading of the executable code on H;
- creation of the BB-manager, as an instance of the corresponding object template in T: the initialization phase of the BB-manager performs the configuration phase of the BB;
- return of the interface reference of the BB-manager.

An object inside a BB instance can create new instances of the object templates included in the corresponding BB template. Object creation requests should be provided with:

- the name of the object template;
- the list of actual parameters, used to initialize the instance.

The following actions are performed:

- internal configuration of the object and activation of its interfaces;
- execution of the initialization phase specified in the template;
- return of the result of the initialization phase and the reference to the main interface.

When an object terminates its initialization phase, it enters an idle state, waiting for the requests of services to its interfaces. When a request is received it is served by the object (according to the described behaviour and the synchronization constraints).

*Interaction constructs*

An object (the client) can interact with another object (the server) through an interface provided by the server. If the two objects are inside two different BBs the interface must be a contract. There are two kinds of interactions:

- synchronous: interactions with a request phase and a result phase (that could be just a notification of termination);
- asynchronous: interactions with only a request phase.

There are two modalities to perform a synchronous interaction:

- blocking: after the request, the client suspends, by waiting for the result;
- deferred-blocking: after the request, the client can go on with internal processing up to a point where it explicitly blocks, by waiting for the result.

A deferred-blocking request is identified by a voucher, used to wait for the result. Interactions are used to:

- request to execute an operation: the invocation of an operation with a result (resp. without a result) is a synchronous (resp. asynchronous) interaction corresponding to an ODP interrogation (resp. announcement); when the server receives such a request it executes the methods associated to the operation in the interface declaration;
- request to access (to read/write) an attribute: the interaction is synchronous; when the server receives such a request it reads/writes the corresponding state variable (the object template can redefine these "default" methods, with other user-defined ones);
- express the interest to receive a notification: the interaction is synchronous; when a client wants to receive a particular notification emitted by an object it must inform it of its interest, and must indicate the method that must be executed when these notifications are received (the so-called notification handler).

When a server emits a notification, the notification is sent to all the "interested" objects, that will handle it through the corresponding notification handler; the server does not wait for any answer from the "interested" objects. A notification emission corresponds to a set of ODP announcements.

The notification handler must have a type compatible with the type of the notifications that they must deal with (same number of arguments, same types).

*Concurrency control*

The model provides two kinds of concurrency:

- inter-object concurrency: there are several objects "running" at the same time;
- intra-object concurrency: there are more execution threads in a single object to allow the concurrent execution of methods; this property must be declared in the object templates.

The model introduces several mechanisms to control and synchronize concurrent activities:

- guards: they are boolean expressions (predicates on the values of the state variables) associated to methods in object templates: the execution of a method invocation is delayed if the corresponding guard is false;
- semaphores: they allow a fine-grain control on accesses to variables, to guarantee the consistency of the object state;
- transactions: they guarantee ACID properties to computations spanning several objects.

The transaction model supported is based on explicit transaction identifiers that identify the transactional activities.

The model provides unstructured primitives for initiation, termination with success or with failure of transactions: the initiation primitive returns a fresh transaction identifier that must be used to include the execution of actions within the scope of the transaction.

The actions that can be submitted to a transaction are:

- invocation of an operation accordingly defined "transactional"; for asynchronous operations only their actual delivery to the server is under the scope of the transaction (according to the hold-until-commit semantics);
- assignment of a state variable: if the transaction aborts the infrastructure assigns to this variable the previous value;
- lock of a variable: to force ordering on the execution of transactions on the same data.

The default transactional model is "flat": any transactional operation invocation is executed under the scope of the current transaction. The model also supports an explicit creation of subtransactions.

*Access to architectural services*

The model can be enriched with additional architectural services together with the related primitives. One of these architectural services is the Trader: the Trader can be accessed to import, export and delete interface references and to manage its internal structures (see the functionalities of the Trader in ANSA [4]).

## 3. THE ODIN PLATFORM

ODIN (Object Distributed INterfaces) is a set of constructs to program "in-the-large" distributed systems according to the described model. The constructs are thought to be complemented by behavioural parts written in C. It consists of two parts:

- a set of templates to define interfaces, objects and virtual nodes (a sort of primitive BBs);
- a C library, forming an API for the behavioural primitives of the model.

The current version of ODIN supports the following features in the model:

- BBs, that can be easily coded through the virtual nodes;
- creation and termination of virtual nodes and objects;
- guards to enable method execution in objects without intra-object concurrency;
- interactions: invocation of synchronous and asynchronous operations; deferred-blocking invocation of synchronous operations through the introduction of vouchers (similar to ANSAware); access to attributes, request, emission, handling of notifications;
- trading functionalities.

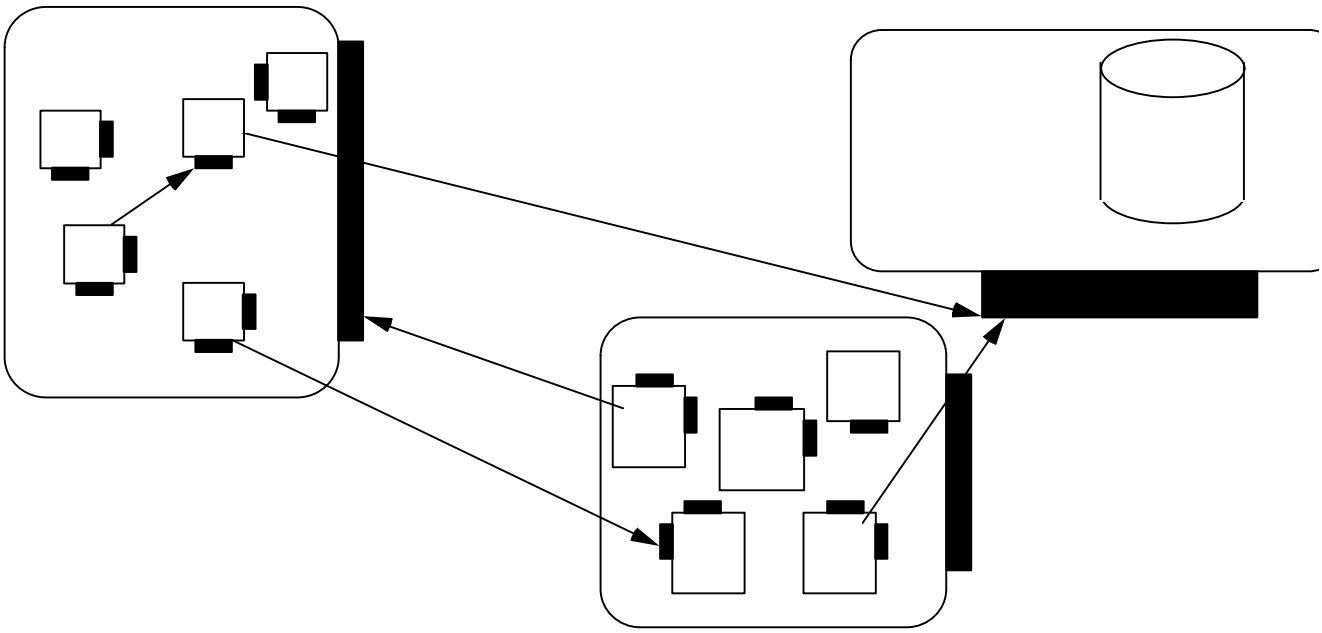
Other functionalities are in course of introduction:

- deferred-blocking mode for all the primitives with a synchronous semantics;
- synchronization primitives for objects that support intra-object concurrency;
- transaction support.

At the moment no type-checking facilities are provided at the ODIN level. They will be introduced at a higher level, i.e. at the level of a more abstract linguistic layer.

A definition of a system in ODIN consists of:

- a set of interface type declarations;
- a set of templates of objects, servers and clients of services provided through interfaces of the declared types;
- a set of templates of virtual nodes, which group object templates in software packages that can be independently loaded/instantiated.



At run-time a system is composed by:

- a Trader;
- a (dynamic) set of virtual node instances, that are units of creation/execution of objects;
- a set of programs, that initialize the system.

### 3.1. Virtual nodes

An ODIN virtual node is defined by a named template which lists the object templates whose instances can be created/executed.

```
TEMPLATE:  Bank
       WITH ATM
       WITH Account
       WITH ...
```

In a system, a template can have several instantiations (possibly on the same host), each of which is identified by a logical name and by an interface that provides the following services:

- instantiation of one of the object templates listed in the corresponding virtual node template;
- termination of the virtual node (along with the destruction of all the contained objects).

The following construct creates a new instance of the template `T`, on the host `H`, identified by the logical name `L`.

```
retcode = ODIN_CreateVN("H","T","L",&VNref);
```

The reference to the virtual node interface, returned by the creation construct, is automatically exported to the trader, with the properties: `name = L; node = H; template = T`.

The following construct terminates the virtual node whose interface is referenced by VNref:

```
retcode = ODIN_DestroyVN(VNref);
```

The above constructs have a synchronous blocking semantics.

### 3.2. Objects

The ODIN objects provide a (static) set of multiple interfaces. The objects are created by instantiating parametric object templates that define their structure:

- the formal parameters, used to initialize single instances;
- the kind of supported concurrency;
- the state variables;
- the interfaces provided by the object (at least the main one): the declaration of an interface specifies the its type and for each operation the method to be executed when invoked;
- the notification handlers, to deal with the notification received upon request: a declaration specifies the type of the handled notification and the method to be executed when a notification is received;
- the behaviour, expressed in C language:
  - the initialization and termination phases (C statements);
  - the methods associated to the operations provided by the interfaces and to the notification handlers (C functions);
  - the guards associated to the methods (C functions);

```
TEMPLATE: Account SIGNATURE: [ Ini_found: currency_t ]
        CONURRENT no
        STATE Cash : currency_t
        INTERFACE DEFAULT Access : Client [Drawing -> M_Dra ,Deposit -> M_Dep ]
        M_Prel  WHEN  Check_Dra
        BEGIN_INIT Cash = Ini_found;  END_INIT
        BEGIN_TERM                END_TERM
        BEGIN_IMPLEMENTATION
                int M_Dra(currency_t amount, currency_t *rest) {
                        *rest = Cash - amount;
                        if(*rest > 0)        {Cash -= amount; return AP_SUCCESS;
}
                        else                 return AP_FAILURE; }
                int  M_Dep(currency_t  amount)  {  Cash  +=  amount;  return
AP_SUCCESS; }
                Bool  Check_Dra()  {  if(Cash>0)  return  APtrue;  else  return
APfalse; }
        END_IMPLEMENTATION
```

*Object manipulation*

The primitive to create an instance of an object template must specify the virtual node where the instance has to be allocated (by using the corresponding interface reference): the object template must occur in the template of the selected virtual node. The creation construct also specifies, besides the template, the actual parameters and the placeholders for the results of the initialization phase and for the main interface reference of the newly created object.

```
retcode = ODIN_Create_ObjTemplate(VNref,a 1,...,a n,&r 1,...,&r m,&ObjIfref);
```

When an object is created, after the execution of its initialization phase, it waits for requests to be served by its interfaces and notifications to be dealt with by its notification handlers.

The following construct terminates, after the execution of its termination phase, the object whose main interface is referenced by `ObjIfref` (of type `TypeName` ).

```
retcode = TypeName_ODIN_Destroy(ObjIfref);
```

These constructs have a synchronous semantics and are called in blocking mode.

*Invocation of operations*

ODIN supports both interrogations and announcements. The following construct causes the invocation of the operation `OpName` on an interface of type `TypeName` referenced by `Ifref`:

```
retcode = TypeName_OpName(Ifref,a 1,...,a n,&r 1,...,&r m);
```

If the operation returns a result (which can possibly be empty), it is invoked in a blocking mode. In order to invoke it in a deferred-blocking mode, the following two constructs must be used, by relating their invocations through a variable `v` of type `ODIN_voucher`:

```
v = TypeName_OpName_Fork(Ifref,a 1,...,a n);
.....
retcode = TypeName_OpName_Join(v,&r 1,...,&r m);
```

These constructs are similar to the ones adopted in the static access mode of CORBA [5]. When the object that provides the interface referenced by `Ifref` receives the invocation, it tries to execute the method associated to `OpName`. In the objects that do not support internal concurrency the execution of the methods is serialized, but the actual execution is delayed if the guard is false.

*Other interaction primitives*

ODIN supports attribute accesses (with a synchronous blocking semantics) and notifications. The following construct reads/writes the current value of the attribute `AttrName` made visible by the interface reference by `Ifref` of type `TypeName`:

```
retcode = TypeName_Get_AttrName(Ifref,&curr_value);
retcode = TypeName_Set_AttrName(Ifref,new_value);
```

When the object which provides the interface referenced by `Ifref`, whose local name is `IfName`, receives the invocation, it reads/writes the state variable `IfName_AttrName`.

The following constructs allow the interactions through notifications. A client can register its interest to the notification `NotifName` emitted by the interface referenced by `Ifref` of type `TypeName` with the call:

```
retcode = TypeName_ODIN_Request(Ifref,TypeName_NotifName,HandlerName);
```

where `HandlerName` is the name of one of the handlers declared in the client template. When the client receives such a notification, it executes the method associated to `HandlerName` (when it is enabled). The client can undo this request with the call:

```
retcode = TypeName_ODIN_UndoRequest(Ifref,TypeName_NotifName);
```

These constructs have a synchronous blocking semantics. An object emits a notification `NotifName` through `IfName`, one of the interfaces declared in its template, of type `TypeName`, with the following call:

```
ODIN_Emit_TypeName_NotifName(IfName,a 1,...,a n);
```

which sends to all the interested object the notification `NotifName(a 1,...,a n)`. The emitting object does not wait any answer and continues the execution without any suspension.

*Access to the Trader service*

ODIN provides a Trader service similar to the basic one in ANSAware:

- export of an interface referenced by Ifref of type `TypeName` , in the context `Context` and with the property list `PropList` :

```
retcode = ODIN_Export(Ifref,TypeName,Context,PropList);
```

- import of an interface of type `TypeName` , in the context `Context` and with a property list that fulfils the constraints `PropConstraints` :

```
retcode = ODIN_Import(TypeName,   Context,PropConstraints,&Ifref);
```

- withdrawing of an interface referenced by `Ifref` with a property list that fulfils the constraints `PropConstraints` :

```
retcode = ODIN_Withdraw(Ifref,PropConstraints);
```

We report some C-statements of a possible method written in ODIN:

```
retcode1 = ODIN_CreateVN("xyz","Bank","XYZ_23",&VNref);
retcode2 = ODIN_Import("VNType","/VNCntx","name = XYZ_23",&Bref);
retcode2 = ODIN_Create_Account(Bref,1000,&ACref);
retcode3 = Client_Drawing(ACref,400,&x);
```

### 3.3. Interface types

An interface type defines the services provided by an interface:

- the operations: the name, the signature and the invocation mode (synchronous/ asynchronous);
- the attributes: the name, the type and the access modes (read or read&write);
- the notifications: the name and the argument types.

In addition in an interface type it is possible to define structured data types. The interface type can be defined as an extension/subtype of other interface types.

```
Client : INTERFACE
         currency_t  : TYPE  =  alias(ap_Integer_t) ;
         Drawing  : SYNCHRONOUS  [amount: currency_t ] RETURNS [currency_t ];
         Deposit  : ASYNCHRONOUS [amount: currency_t ];
```

## 4. IMPLEMENTATION OF ODIN ON DCE/C++

The current implementation of ODIN is based on a DCE/C++ platform. The main rationale at the basis of DCE's choice is that OSF/DCE [6] is becoming a widely accepted industry product, and it's available for many hardware platforms; furthermore, it simplifies the transition to a transactional version of ODIN (through DCE-based Transarc/Encina [7]). The usage of C++ [8] enables a smooth interfacing among the distributed object model and a variety of (available or under development) OO software , including OODBMS's.

The implementation is based on the following mappings:

- the virtual nodes are translated into DCE servers;
- the object templates are translated into C++ classes, while an ODIN object becomes a C++ objects (running in a DCE server);
- the ODIN interfaces are translated into DCE interfaces;
- the references to ODIN interfaces are C structures that contain information on how an interface may be reached.

The following sections briefly describe the main aspects of this implementation (the aspects concerning subtyping, attributes, notifications, object creation/termination are not covered).

### 4.1. Coding of interface references

An interface reference (provided by an object *obj* running on a virtual node *vn*) contains information on how an interface may be reached by objects within a system, which may be grouped in three distinct fields:

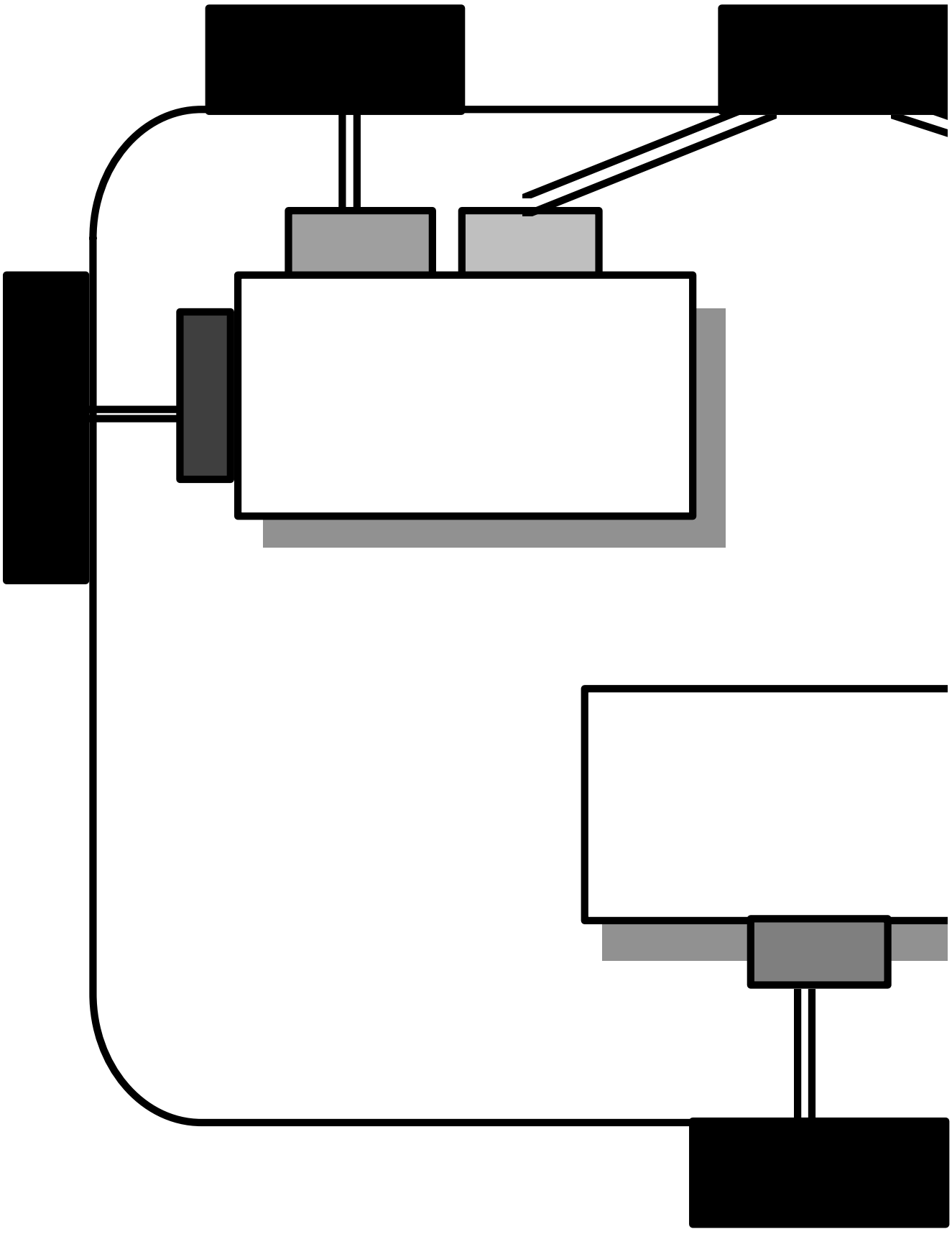
1) *how to reach vn*: protocol used + internet address of the host + port number on which is listening the server corresponding to *vn*;
2) *how to reach obj* the pointer to the C++ object instance corresponding to *obj*;
3) *how to reach the interface*: an integer that identifies the interface among the others provided by the same object.

### 4.2. The structure of a DCE server

A virtual node template is transformed into the executable code of a DCE server. The DCE server provides an environment to create, invoke and run the C++ objects corresponding to the objects in the virtual node instance. Such a DCE server provides the union of the interfaces of the objects supported by the corresponding virtual node (in addition to some service interfaces to perform object creation and to handle notifications) and its behaviour consists on a set of C++ classes corresponding to the objects templates listed in the virtual node template.

The main components in such a server are:

- the DCE stub routines for the interfaces provided by the server;
- the C++ classes corresponding to the object templates supported by the virtual node;
- ODIN stub routines to perform invocations of methods on C++ objects.



One of the main task of the server is to call the correct method on the correct C++ object when it receives an operation invocation on one of its interfaces. In fact, an interface (instance of a type T) provided by a server is shared by all the object instances that provide an interface of type T (i.e all the instances of a template share the same interfaces): this could be intended that the server has multiple implementations (called in DCE manager routines) of a single interface. In our implementation we haven't used DCE's object: we manage ourselves the object and method selection through the field (2) and (3) of the interface reference structure.

### 4.3. From ODIN interfaces to DCE interfaces

An ODIN interface is implemented as a DCE's interface: the operations and the attribute declarations are coded as DCE operations. The signature of each operation defined in the DCE IDL is slightly different from the one specified in the ODIN interface template: for each operation, the data to reach the destination interface is added (i.e., the C++ object pointer and the interface identifier within the object). Some auxiliary operations are added to cope with object termination and notification registration. The types declarations in the ODIN interface templates are transformed in the corresponding DCE/IDL type declarations.

### 4.4. From DCE operation invocation to C++ method call

As mentioned before, one of the main task of the server is to call the correct method on the correct C++ object when it receives an operation invocation on one of its interfaces. This selection (for a generic operation `Op` of an interface of type `Type`) is performed through a 4 layer code structure:

- *server stub code*: this code (automatically generated by the DCE starting from the IDL definition of an interface) is by a fresh thread generated by the DCE infrastructure when a rpc call arrives to the server; this stub performs the unmarshalling the arguments, the invocation of the manager code and the marshalling of the results;
- *manager code*: this C code invokes the `bridge_Type_Op()` , passing to it the C++ object pointer and the interface identifier, the arguments and the pointers to the results;
- *bridge code*: this C++ code performs a first switching among the possible C++ classes which have at least one `Type` interface; it uses the C++ object pointer to select the right object instance, whose class member `_ODIN_Type_Op()` function is invoked, passing to it the interface identifier, the arguments and the pointers to the results;
- *object code*: the last switching concerning the selection of the interface (an object may have more than one interface of the same type) is performed by `_ODIN_Type_Op()` that call the right C++ method, according to the operation/method association in the object template.

In the classes that do not support intra-object concurrency, the routine `_ODIN_Type_Op()` performs concurrency controls, i.e. serialization of method executions and guards evaluation:

- before invoking the selected method, the routine checks if the method is enabled: if it is not enabled (i.e. either the object is executing another method or the guard evaluate to false), the thread is blocked (on a DCE mutex/semaphore) and wait for the completion of the execution of the previous one;
- after invoking the method, the routine checks if there is a suspended method that has become enabled and (if any) resumes the corresponding thread.

The invocation of a generic operation `Op` of an interface of type `Type` (having as arguments the interface reference of the server interface and the other arguments of the call) at the client side is much simpler, through the following layers:

- *call layer*: this code hides to the caller the specific implementation (an operation invocation performed by an object's method must follow the API specs: it is the same independently of the platform, DCE/C++, on which it's built on); this layer invoke the rpc operation `Op()`;
- *client stub code*: this DCE generated code marshals the arguments, does the rpc and unmarshals the results.

If the server interface is in the same virtual node of the client, the "call layer" performs a shortcut, and call the corresponding "bridge" routine within the same virtual node.

Different routines in the call layer implement the different kinds of invocations (blocking synchronous, deferred/non-blocking synchronous and asynchronous):

- blocking synchronous: the calling thread performs a rpc which keeps it blocked until the answer from the server is got;
- deferred/non-blocking synchronous: the calling thread spawns a new thread that performs the rpc; the original thread, when it wants collecting the results, joins the spawned one and gets the result;
- asynchronous: the calling thread spawns a new thread that performs the rpc, and then cancels itself.

### 4.5. From ODIN object templates to C++ classes

A C++ class is generated for each ODIN object template. The private state variables of this C++ class can be grouped in the following sets:

- the state variables in the ODIN template;
- the interfaces defined in the template;
- the internal data structure for concurrency control, registration of notifications, etc.

All the functions defined in the implementation part of the template become private methods of the C++ class. The public methods of the class are:

- the Init method invoked during the initialization phase;
- the methods invoked by the bridge code (the `ODIN_<Type>_Op()` functions);
- the class constructor that performs internal initializations;
- the class destroyer that performs the termination phase.

In the following the structure of a C++ class is outlined. The object is named "A", and it has three interfaces: X, Y (with type T1) and Z (with type T2).

```
File: A.h ------------
class A: public OdinRoot {
  ...                       // internal data structure for concurrency control
  odin_Ifkey_t X;           // T1                        // interface variables
  odin_Ifkey_t Y;           // T1
  odin_Ifkey_t Z;           // T2
  odin_Integer_t SomeState;                                    //     object     state
variables
  odin_Cardinal_t AnotherState;
  // BEGIN_IMPLEMENTATION
  int Meth_11x(...) {...};         int Meth_11y(...) {...};    int        Meth_12x(...)
{...};
  int Meth_12y(...) {...};         int Meth_21(...) {...};
  // END_IMPLEMENTATION
  public:     Init(...);    A(...);       ~A(...);
  int _ODIN_T1_Op11(...);   int _ODIN_T1_Op12(...);     // Interface T1
```

```
  int _ODIN_T2_Op21(...);  };                              // Interface T 2

File: A.C ------------
A::Init(...)  {// INIT user code } A::A(...) {...} A::~A(...) {// TERM user code
...}
int _ODIN_T1_Op11(...) {                                   // Interface T1: Op11
  int res = ODIN_SUCCESS;
  if(ifid == X)      {_ODIN_In(ODIN_METH_Meth_11x);res       =       Meth_11x(...);
_ODIN_Out();}
  elseif(ifid == Y)         {_ODIN_In(ODIN_METH_Meth_11y);res    =    Meth_11y(...);
_ODIN_Out(); }
  else               res = ODIN_FAILURE;
  return res; }
int _ODIN_T1_Op12(...) {                                   // Interface T1: Op12
  int res = ODIN_SUCCESS;
  if(ifid == X)      {     _OD IN_In(ODIN_METH_Meth_12x);res     =     Meth_12x(...);
_ODIN_Out(); }
  elseif(ifid == Y)         {  _ODIN_In(ODIN_METH_Meth_12y);  res  =  Meth_12y(...);
_ODIN_Out();}
  else res = ODIN_FAILURE;
  return res; }
int _ODIN_T2_Op11(...) {                                   // Interface T2: Op21
  int res = ODIN_SUCCESS;
  if(ifid == Z)      {    _ODIN_In(ODIN_METH_Meth_21);    res    =    Meth_21(...);
_ODIN_Out(); }
  else res = ODIN_FAILURE;
  return res; }
```

Some remarks have to be done:

- the `_ODIN_In/_ODIN_Out` routines control the concurrency and method enabling: these are not present in the objects supporting intra-object concurrency;
- the class OdinRoot is a superclass for all the classes in ODIN: it contains specific methods and information shared among all object classes.

## 5. PERFORMANCE

The rough performance of the implementation was measured w.r.t. elapsed time on a IBM RISC 6000 slightly loaded, using the UDP/IP protocol.

The most important figure is the time to perform an object-based RPC: it is around 14 ms for an RPC with null body. This quite high time is due to the fact that it comprises also the time for converting (through a DCE routine) a binding handle representation from an external (character-string) form to the internal one: without this transformation object-based RPC is equivalent to normal DCE RPC. Also distributed object creation is comparable with an RPC (plus the negligible creation time of a C++ object).

## 6. FUTURE WORKS

In the current implementation some open issues still remains. The most important are:

- to introduce some kind of persistent objects: this could be done by integrating an OODB into the existing platform;
- to add some transactional capabilities to the implementation, specifying which are the operations and the chain of invocations which must be covered by a transactional umbrella: in doing this, the main support could come by using an already existing transactional kit (Transarc/Encina) which is built upon DCE;

- to add constructs to control intra-object concurrency.

## REFERENCES


1. ISO/ODP, Basic Reference Model of ODP: prescriptive model, ISO/IEC JTC1/SC21 N7525 (dec. 1992).
2. Bellcore, INA Cycle 1 Framework Architecture, Issue 1.5, TM-NWT-020237, (dec. 1991).
3. P.G. Bosco, G. Giandonato and C. Moiso, A distributed processing model for telecommunication services and operations software, in Proc. TINA'93 (Sept. 1993).
4. APM Limited, ANSAware 3.0 Implementation Manual (Jan. 1991).
5. OMG, The common object request broker: architecture and specification, OMG Document Number 91.8.1 (Aug. 1991).
6. OSF/DCE, Introduction to OSF DCE (Prentice Hall, 1992).
7. Transarc Corporation, ENCINA, Transarc Corporation (1991).
8. B. Stroustrup, The $C^{++}$ Programming Language (Second Edition), Addison-Wesley Publishing Company (1991).