A distributed object-oriented model for telecommunication services

P.G. Bosco, G. Giandonato, G. Martini, C. Moiso
CSELT
Via Reiss Romoli 274, 10148 Torino, Italy

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 Reference Model, [ODP92]. In particular we concentrate here on the computational and engineering viewpoints.

The motivation for this research comes directly from the Operating Company needs of mastering the software complexity imposed by rapidly growing and highly diversified market demands, and consequently, by the necessity of rapid and economical development of new services.

The approach we are investigating to achieve such an ambitious, yet strategic to the Network Operators, goal, is based on a single conceptual model - rich enough to express the descriptive, processing and deployment needs of different areas like advanced Intelligent Networks, TMN and Operation Systems - where concepts of different areas have been unified. 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 Reference Model, 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 [INA91], are reported in [Bosco93].

Building-blocks

A distributed application consists of one or more BBs, each installed on a single host node. 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 BB 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 other management functionalities.
The objects inside a BB 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.

Objects

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

- 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.

An object can provide a statically defined set of different interfaces, one of which is elected as the main object interface.

Interfaces

An interface provides to its clients:

- **operations**: to invoke methods of the object;
- **attributes**: to read/write state variables of the object;
- **notifications**: to require to be noticed 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/contracts are denotable values that can be stored in variables and passed as arguments/results when interactions are performed.

2.2 Typing

The model is equipped with a type system (which extends the type system of the embedded languages used to describe object behaviour with types which are specific to the model: interfaces, main interfaces, operations, attributes,...) that guarantees the correctness of the interactions among the objects: in a well-typed program there are no operation requests or

notifications to (dynamically determined) interfaces that are not able to handle them, as well as no accesses to non-visible attributes. Compatibility of newer software packages (BB's) in an evolving distributed system is guaranteed by the introduction of a general 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 dealt with by the model can be grouped in the following way:

- constructs to manipulate BBs 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).

Instance manipulation

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 should (at least) specify:

- 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 of a template T is created on a 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 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.

Two kinds of interactions are available:

- **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.

and 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, afterwards, to wait for the result.

Interactions are used for the following purposes:

- request to execute an operation: the invocation of an operation with a result (resp. without a result) is a synchronous (resp. asyncronous) 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);
- to express the interest to receive a notification.

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 level 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 and invocation parameters) 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 the explicit handling of transaction identifiers. It provides unstructured primitives for initiation, termination with success (commit) or with failure (abort) 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 (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 nested transactions.

Access to architectural services

The model can be enriched with additional architectural services together with suitable modalities to access such services. In some cases this could be realized by "objectifying" architectural services and defining the appropriate contracts. In other cases (or, at another level of abstraction) there could be predefined behavioural constructs to make more evident the use of such services. Examples of architectural services are the **Trader**, and the **Notification Handler**.
The Trader can be assessed to import, export and delete interface references and to manage its internal structures (see the functionalities of the Trader in ANSA, [ANSA91]).

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 meant to be complemented by behavioural parts written in C.

ODIN consists of two parts:

- a **set of templates** to define interface types, 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:

- BBs, that can be easily coded through the virtual nodes;
- creation and termination of virtual nodes and objects;
- guards to enable method execution in sequential objects;
- 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;
- flat transactions on sequential objects;

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;
- full transaction support;

At the moment no type-checking facilities are provided at the ODIN level. They will be present at the level of a more abstract linguistic layer, offered by an integrated CASE environment.

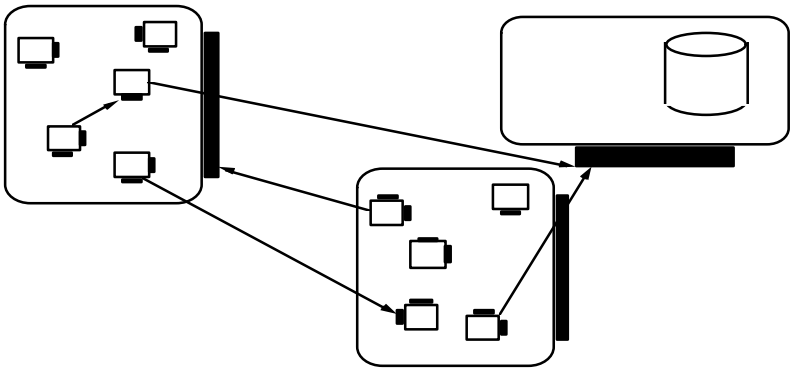
Structure of an ODIN application

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 template 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 `template name` , on the host `host name`, identified by the logical name `logical name` .

```
retcode = ODIN_CreateVN("host name","template name","logical name",&VNref);
```

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

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

Accordingly to the model, 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 state variables;
- the interfaces provided by the object (at least the main one): an interface declaration specifies the type of the interface 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, i.e.:
  - 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   ]

      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
```

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 terminate the object whose main interface is referenced by `ObjIfref` (of type `TypeName`); the object is destroyed after the execution of the termination phase:

```
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_O pName_Join(v,&r 1,...,&r m);
```

These constructs are similar to the ones adopted in the static access mode of CORBA [OMG91]. 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.

Attributes

ODIN supports access to object attributes (with a synchronous blocking semantics). 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` .

Notifications

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_Un doRequest(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 suspending.

Access to the Trader service

ODIN provides a trading service similar to the ANSAware Trader:

- 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);
```

3.3 Interface types

An interface type defines the services provided by an interface, i.e.:

- 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 new structured data types.
By means of an inheritance construct, an interface type can be defined as an extension/subtype of other interface types.

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

4. Platform Implementation

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 [OSF92] 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 [Trans91]). The usage of

C++ [Strou91] enables a smooth interfacing among the distributed object model and a variety of (available or under development) OO SW , including OODBMS's.

The implementation is based on the following mappings:

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

6. Conclusions

In the currently available implementation some open issues remain. The most important are:

- to define some kind of "persistent" property, to be added to some 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 synchronize executions within an object with intra-object concurrency;
- to extend to microkernel-based, real-time operating systems.

Nevertheless the core concepts of the model are sufficiently stable and the implementation robust enough to base on it large-scale integration experiments of the TINA, [GGAP93], architecture, e.g. those described in [BMG92].

References

[ANSA91] **APM Limited**, ANSAware 3.0 Implementation Manual (jan. 1991).

[BMG92] G. P. Balboni, G. Giandonato, R. Melen
Experimenting TINA-oriented signalling and control on a B-ISDN switch prototype, Proc. Tina '92, Narita, Japan (jan. 1992)

[Bosco93] **P.G. Bosco, G. Giandonato, C. Moiso**
A distributed processing model for telecommunication services and operations software, Proceedings TINA'93 (sept. 1993).

[GGAP93] G. Giandonato, A. Pelaggi
TINA, network architectures and system architectures
In these Proceedings

[OSF92] OSF**,** Introduction to OSF DCE (Prentice Hall, 1992).

[INA91] **Bellcore**, INA Cycle 1 Framework Architecture, Issue 1.5, TM-NWT-020237, (dec. 1991).

[ODP92] **ISO/ODP**, Basic Reference Model of ODP: prescriptive model, ISO/IEC JTC1/SC21 N7525 (dec. 1992).

[OMG91] **OMG**, The common object request broker: architecture and specification, OMG Document Number 91.8.1 (aug. 1991).

[Strou91] **B. Stroustrup,** The C++ Programming Language (Second Edition), Addison-Wesley Publishing Company (1991).

[Trans91] **Transarc Corporation,** ENCINA, Transarc Corporation (1991).