An object-oriented distributed transactional platform supporting a TINA-like computational model

Pier Giorgio Bosco, Ennio Grasso, Giovanni Martini, Corrado Moiso

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

1. INTRODUCTION

The purpose of this paper is to present some preliminary results of a research activity carried on at CSELT on the prototype implementation and definition of a distributed processing environment, based on the ODP and TINA-C models [1,3]. This work extends the results of a previous activity [2], by implementing an extended transaction model.
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 model is given in Sect. 2. The remaining part of the paper is focused on a specific realization of the model named ODIN (Object Distributed INterface),which consists of a C-API (Sect. 3) and of its implementation (Sect. 4) by means of a widely available software platform (OSF/DCE, Transarc/Encina and C++).

2. THE DISTRIBUTED PROCESSING MODEL

The model, based on the object-oriented paradigm, 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.

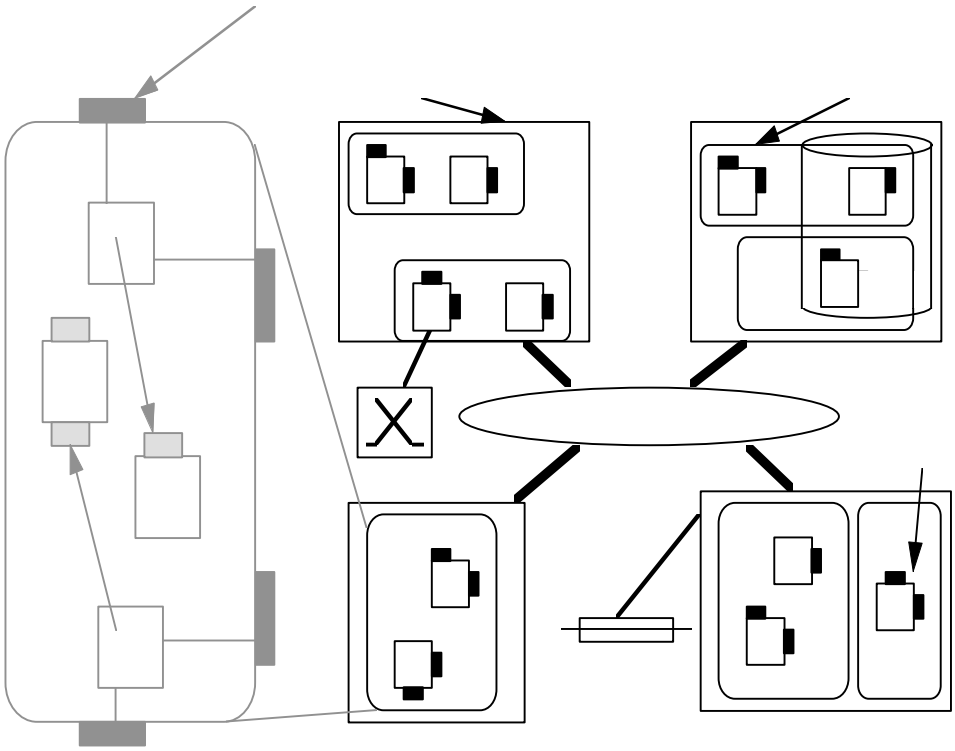
2.1. Structural aspects

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 (BBs), 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 TINA-C initiative [3], are reported in [4].

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 whose internal state and behaviour is described by means of a set of object templates (one of which is elected as BB-manager) 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.



The structure and the behaviour of the objects in a BB are described by (parametric) object templates. The objects are units of functionality and encapsulation that can interact through interfaces. An object template includes the following fields:
- 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 (which corresponds to the object reference). An interface provides to its clients the same kind of services provided by OSI managed objects [5]:
- operations: to invoke methods of the object by mapping the former onto the latter;
- attributes: to read/write state variables of the object;
- notifications: to be notified of something happened in the object.

The model can be enriched with additional *architectural services*, implemented by system BBs. An example of these architectural services is the Trader [1].

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 nor notifications to

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

The type system introduces a subtype relation among the interfaces: an interface type is a subtype of another one if it provides more operations, attributes and notifications.

2.3. Behavioural aspects

The behavioural aspects considered by the model can be grouped in constructs to:
- manipulate instances of BB and objects (creation, termination,...);
- perform interactions between objects;
- control the concurrent activities;
- perform transactions.

Manipulation of instances
At run-time, BBs and objects can be dynamically created as instances of templates.
To create a new BB instance it is necessary to provide the name of the BB template, the actual parameters and the name of the host where to install the BB. The system creates on the target host the BB-manager, whose initialization phase performs the configuration of the BB: the creating object receives the BB-manager main interface reference, that will be used as reference to the BB as a whole.

An object can instantiate in its BB the object templates included in the corresponding BB template. After executing the initialization phase, the new object returns the result of the initialization phase and its main interface and waits for clients to request services through its interfaces.

Interaction constructs
An object (the client) can interact with another object (the server) through an interface provided by the server. There are two kinds of interactions:
- synchronous: with a request and a result phase;
- asynchronous: 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 continue the internal processing up to a point where it explicitly blocks and waits for the result.

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

Interactions serve to:
- request the execution of an operation: the invocation of an operation with a result (resp. without a result) is a synchronous (resp. asynchronous) interaction; the operations with a result are called interrogations in the ODP terminology, while those without results are called announcements; when the server receives the request executes the method associated to the operation in its interface declaration;
- request the access (read/write) of an attribute: the interaction is synchronous; when the server receives such a request it reads/writes the corresponding state variable;
- express the interest to receive a notification: the interaction is synchronous; a client wishing to receive a particular notification emitted by an object must inform the object of its interest, by

indicating the method (notification handler) that must be executed when the notifications are received.

A notification, emitted by an object, is sent to all the "interested" clients, which will handle it through the corresponding notification handler; the object does not wait for any answer from the "interested" objects.

Concurrency control
The model provides inter-object concurrency and intra-object concurrency (there are more execution threads in a single object). The model introduces several mechanisms to control and synchronize concurrent activities:

- guards: predicates on the values of the state variables associated with methods in object templates; the execution of a method invocation is delayed until the its guard is true;
- locks: allow a fine-grain control on thread synchronization; if the lock-request is issued on behalf of a transaction, the transaction itself becomes the owner of the lock. Conversely, if the thread is not associated with a transaction, the lock behaves like a semaphore and must be explicitly released by the thread;
- open/close nested transaction.

Transactional behaviour
The model supports the nested transaction model [6] on a per object basis. There are no primitives to explicitly begin transactions: they are implicitly started by invoking transactional operations/ interrogation. During the execution of a transactional operation an object can invoke other transactional operations. This forces an implicit nesting on the transaction itself giving rise to a tree of subtransactions one for each invoked operation.

If the invocation is issued from inside a transaction, the call spawns a nested transaction in the invoked object. On the other hand, if the invocation is issued from outside transactional boundaries, the call starts a top-level transaction in the invoked object. Transactional operations are always synchronous and return a status code indicating the outcome of the subtransaction. Note that transactions are not identified by explicit identifiers since the transactional context is associated with the thread in charge of the execution of the transactional operation.

Besides nested transactions having full ACID properties, the model also allows the definition of "open" nested transactions [7] so as to relax the ACID paradigm according to the application requirements. In the classic nested model subtransactions transfer all their locks to the parent transaction when they commit. This rule is mandatory to guarantee atomicity and isolation (since committed subtransactions may later be rolled back), but can be detrimental in an environment where transactions may be long-lived. Conversely, open subtransactions release their locks when they commit, thus increasing the overall concurrency degree of the system.

Clearly, the "open" model needs a more complex recovery scheme: undoing an open subtransaction merely by rolling back its work may unintentionally undo the effects of a second interleaved transaction. Hence, abort recovery must be performed by issuing a compensating transaction that *semantically* reverses the corresponding aborted subtransaction. This rule relaxes atomicity since a compensating transaction does not need necessarily to undo all the effects of the transaction compensated for, but can result in a semantic specific repair action.

Since subtransactions are the result of the invocation of transactional operations, each operation with open semantics is associated with two methods: the *primary* and the *inverse* methods; the

latter being invoked (by the transaction manager) if the operation is to be compensated. The inverse method executes as a separate transaction and is supposed to undo the effects of the whole tree whose root is the subtransaction compensated for (no nested compensation is required). The actual parameters of the inverse method are determined during the execution of the primary method.

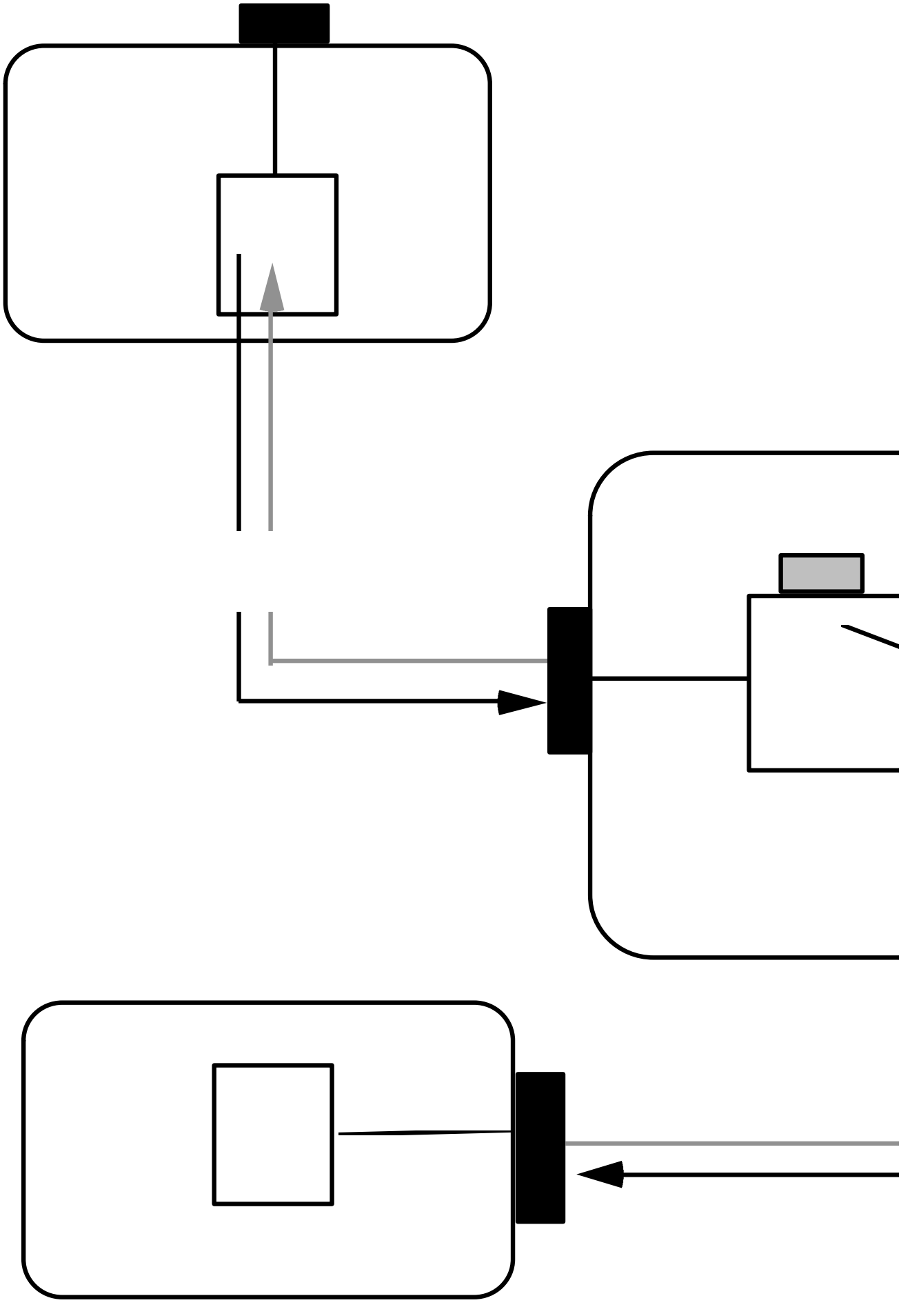
Our transaction model provides a mechanism to improve concurrency by exploiting the semantic knowledge of object operations without jeopardizing serializability [8]. Each object template, declaratively specifies the conflict relationships between the transactional methods based on the application semantics. Two conflicting methods will be serialized, whereas two non-conflicting methods are admitted concurrently (e.g. two `increment` on the same `Counter` object).

The model devises a progammable locking facility as concurrency control protocol. In addition to the usual *shared* and *exclusive* lock-modes, dealing with abstract operations requires specific lock-modes such that two lock-modes are compatible iff their corresponding operations do not conflict. A (sub)transaction performing an operation `Op` on object `Obj` has to acquire an `Op`-*lock* on `Obj` before being executed. Since operation `Op` is performed as subtransaction, the same protocol is applied for the operations invoked by `Op` on other objects.

Transactional creation/termination
The model addresses transactional creation and deletion of objects by exploiting the concurrency mechanism so far described. The implicit statement is that the initialization and the termination methods of an object - respectively executed during creation and termination - conflict with all the other methods (which are protected by an exclusive-lock).

When a transaction creates an object, it acquires the exclusive-lock associated with the initialization method; in this way no transaction other than the creating one can access the transactional services of the object. Then:
- if the creating transaction commits, it releases the exclusive-lock, thus allowing other transactions to access the transactional methods;
- if the creating transaction aborts, the object is destroyed and the waiting transactional requests are sent back with a failure code.

As far as termination is concerned, the model adopts a deferred strategy. First, the *terminate* primitive acquires the exclusive-lock associated with the termination method (on behalf of the terminating transaction). When the transaction manages to obtain the lock the object enters a quiescent state (no requests are served) and the termination method is executed. If the transaction eventually commits, the object is actually terminated; otherwise, it returns operative.



3. THE ODIN PLATFORM

ODIN is a set of constructs to program "in-the-large" distributed systems according to the described model. The constructs are considered complemented by behavioural sections in C/C++. ODIN consists of two parts:

- a set of template forms to define interfaces, objects and BBs;
- a C library, forming an API for the behavioural primitives of the model.

The current version of ODIN supports the following features of the model (by using template fields and library functions):

- creation and termination of BBs and objects;
- interactions: invocation of synchronous and asynchronous operations; deferred-blocking invocation of synchronous operations; access to attributes; request, emission, handling of notifications;
- locks and guards to control interactions and intra-object concurrency;
- transaction support;
- trading functionalities.

The definition of an ODIN system 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 BB templates, 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 BB instances, that are units of creation/execution of objects;
- a set of programs, that initialize the system.

3.1. Building Block

A BB is defined by a named template that lists the object templates whose instances can be created/executed in that BB.

```
BB_TEMPLATE:  Bank
      WITH ATM BB_MANAGER
      WITH Account
      WITH ...
```

At runtime a template can have several instantiations (possibly on the same host), each of which is identified by a logical name.
The following construct creates a new instance (identified by the logical name L) of the template T, on the host H, with optional parameters to be passed to the BB manager:

```
retcode = ODIN_CreateBB_T("H","L",a1,...,an,&r1,...,&rm,&BBMgrRef);
```

The reference to the BB interface `BBMgrRef` (i.e. the main interface of the BB manager) returned by the creation construct is automatically exported to the Trader along with the properties:

```
name = L; node = H; template = T.
```

The following construct terminates the BB whose interface is referenced by `BBMgrRef` :

```
retcode = ODIN_DestroyBB(BBMgrRef);
```

The above constructs have a synchronous blocking semantics.

3.2. Interface types

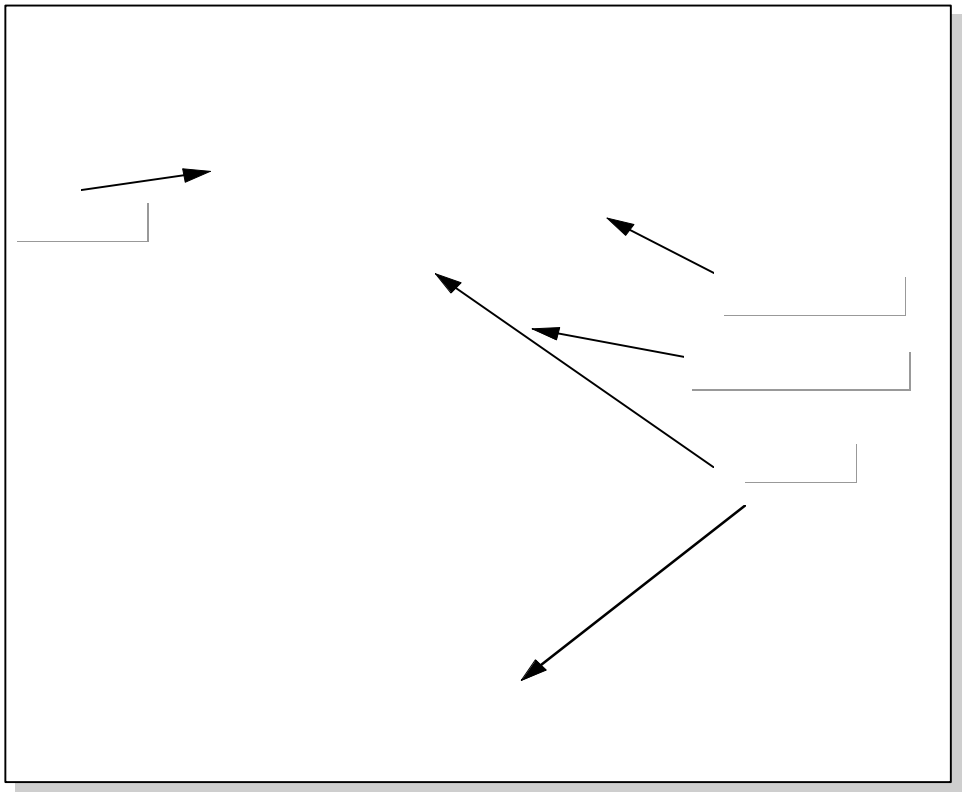
An interface type defines the services provided by an interface:
- the operations: the name,the signature and the invocation mode (synchronous/ asynchronous);
- the attributes: the name, the type and the access modes (read or read&write);
- possible transactional behaviour of operations;
- the notifications: 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
         TRANSACTIONAL
         currency_t  : TYPE  =  alias odin_Integer_t  ;
           Drawing   :   SYNCHRONOUS      [amount:   currency_t  ]   RETURNS
[currency_t ];
         Deposit  : ASYNCHRONOUS [amount: currency_t  ];
```

3.3. Objects

ODIN objects provide a (static) set of multiple interfaces. The objects are created by instantiating parametric object templates defining their structure:
- the formal parameters, used to initialize single instances;
- the state variables;
- the possible transactional behaviour of creation/termination;
- the interfaces provided by the object (at least the main one must be present): the declaration of an interface specifies its type and for each operation the method to be executed when invoked; if an operation has open transaction semantics, the indication of the compensative method has to be added;
- the notification handlers, to deal with notifications received upon request: a declaration specifies the type of the handled notification and the method to be executed as the notification is received;
- the lock conflict table for transactional methods;
- 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).



Object manipulation

An object may create other object instances. The new objects are allocated in the same BB of the creator, so that the instantiated template must occur in the template of the BB. The creation construct 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 created object:

```
retcode = ODIN_Create_ObjTemplate(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 at its interfaces and notifications to be dealt with by its notification handlers.

If the creation is performed on behalf of a transaction, the new object can only serve requests coming from that transaction until commit (if the transaction aborts, the object is destroyed).

Objects cannot destroy other objects, though an object can terminate itself through the construct:

```
retcode = ODIN_Terminate();
```

After the completion of the present activities, the termination phase is executed and the object space is freed.

If the termination is performed by a transaction, the object is not really terminated: no more incoming requests are served until the transaction is resolved. If the transaction commits, the object is actually destroyed, otherwise it returns operative.

Invocation of operations

ODIN supports synchronous/asynchronous operations. 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, which are related by 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 [10]. When the object that provides the interface referenced by `Ifref` receives the invocation, it executes the method associated with `OpName` .
For objects that do not support internal concurrency the execution of methods is serialized.

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 through the interface referenced by `Ifref` of type `TypeName` :

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

When the object providing the interface referenced by `Ifref` receives the invocation, it reads/writes the associated state variable.
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` through the call:

```
retcode = TypeName_ODIN_Request(Ifref,"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 with `HandlerName` (if enabled). The client can undo this request through the call:

```
retcode = TypeName_ODIN_UndoRequest(Ifref,"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` , through 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 continues the execution without waiting any answer.

Transactional primitives
An object can transactionally update its internal state through the call:

```
retcode = ODIN_TransactionalCopy(StateVar,NewStateValue);
```

which must be executed only on behalf of a transaction. If the transaction aborts the state of the variable is restored to the old value (before-image recovery).

A lock of mode `Mode` on the variable `VarName` can be requested through the call

```
retcode = ODIN_LockWait(VarName,Mode);
```

If the lock is requested on behalf of a transaction it will be released at its completion, otherwise it has to be released by calling:

```
retcode = ODIN_LockRelease(VarName);
```

A method invoked in an open-transactional way indicates the parameters `par1,..,parN` to be passed to its inverse method by calling:

```
retcode = ODIN_LogCompensate(par1,...,parN);
```

A transactional methods succeeds invoking the construct `return_with_commit` , whereas it aborts by calling`return_with_abort` .

4. IMPLEMENTATION OF ODIN ON DCE/C++

The current implementation of ODIN is based on DCE/C++. The main rationale at the basis of DCE's choice is that OSF/DCE [11] is becoming a widely accepted industry product, and may support transactional features (adding Transarc/Encina [12]). To manage open-nesting transactions we devised an eXtended Transaction Manager (XTM) laid over Encina/TRAN; since Encina/LOCK provides a static set of lock-modes, we developed a Programmable Lock Manager (PLM) that allows the programmer of each object template to define a conflict table of lock-modes.
The usage of C++ 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:
- BBs are translated into DCE servers;
- object templates are translated into C++ classes, while an ODIN object becomes a C++ object (running in a DCE server);
- ODIN interfaces are translated into DCE interfaces;
- references to ODIN interfaces are C structures that contain information on how the interface may be reached;

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

4.1. Coding of interface references

An interface reference (provided by an object *obj* running on a BB *bb*) contains information on how an interface may be reached and may be grouped in three distinct fields:
1) *how to reach bb*: logical name of *bb*;
2) *how to reach obj*: a logical pointer to the C++ object instance corresponding to *obj*;
3) *how to reach the interface*: an integer that identifies an interface provided by the object.

The use of logical entities eases the installation and reconfiguration of BBs and the adding of persistency capabilities. The runtime system provides the mapping from the logical entities onto the real ones.

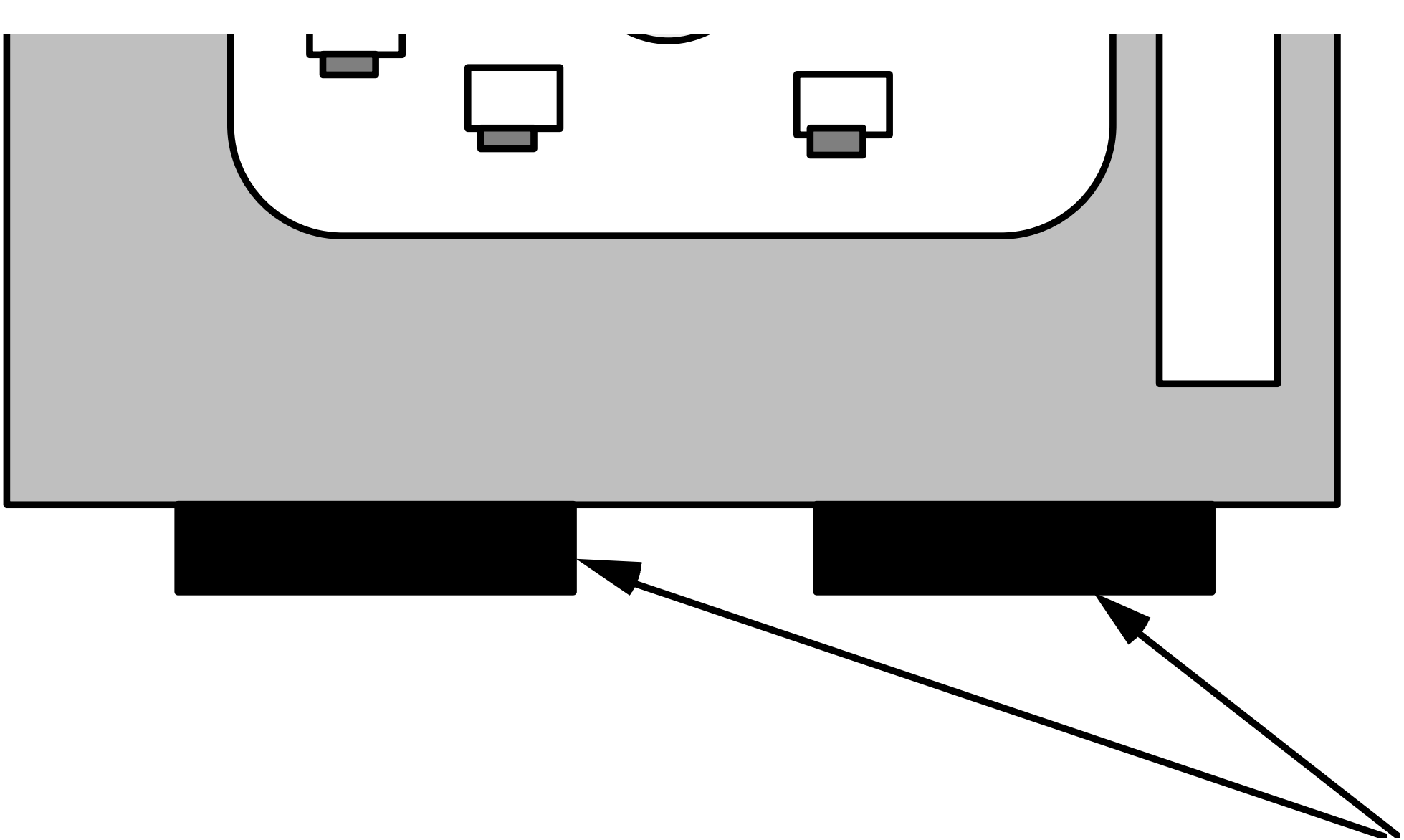
4.2. The structure of a DCE server

A BB template is transformed into the executable code of a DCE server. The DCE server provides an environment to create, invoke and execute the C++ objects corresponding to the

objects in the BB instance. The DCE server provides the union of the interfaces of the object templates supported by the corresponding BB, and its behaviour consists on a set of C ++ classes corresponding to the objects templates listed in the BB 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 BB;
- ODIN stub routines to perform invocations of methods of 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 DCE 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 is to say that the server can have multiple implementations (called manager routines in DCE) of a single interface.



4.3. From ODIN interfaces to DCE interfaces

An ODIN interface is implemented as a DCE interface: the operations and the attribute declarations are coded as DCE operations.
For each operation signature, the information to reach the destination interface is added (i.e., the logical pointer to the object and the interface identifier within the object).
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 specified C++ object instance 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 is generated by compiling the DCE/IDL definition of an interface, and is executed by the DCE infrastructure when a rpc call arrives to the server;
- *manager code*: this C code invokes the `bridge_Type_Op()` , forwarding the C++ logical pointer, 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 an interface `Type`; this code uses the C++ logical pointer to select the right object instance (by looking up an internal mapping table), and the corresponding class member `_ODIN_Type_Op()` function is invoked, passing the interface identifier, the arguments, and the pointers to the results;
- *object code*: the last switching concerns the selection of the interface (an object may have more than one interface of the same type) and is performed by `_ODIN_Type_Op()` . This function calls the right C++ method, in accordance with the operation/method association in the object template; if the invocation is transactional a new subtransaction is spawned.

The function `_ODIN_Type_Op()` performs concurrency controls: serialization of methods' executions and guards' evaluation:

- before invoking the selected method, the routine checks whether the method is enabled: if not (i.e. the guard evaluates to false or the method's lock-mode conflicts with other locks granted to other transactions), the thread is blocked and waits for being resumed;
- 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` at the client side is much simpler, through the following layers:

- *call layer*: it performs the mapping from the logical name of the BB address onto the real one (by asking the Trader to resolve the mapping), and saves the result into an internal cache; then it invokes the rpc operation`Op()`;
- *client stub code*: this DCE generated code marshals the arguments, invokes the rpc and then unmarshals the results.

If the server interface belongs to the same BB as the client does, the "call layer" performs a shortcut by calling the corresponding "bridge" routine within the same BB.

Different routines in the call layer implement 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 reply from the server arrives;
- deferred-blocking synchronous: the calling thread spawns a new thread that performs the rpc; when the calling thread wants to collect the results, "joins" the spawned one and acquires the results;
- asynchronous: the calling thread spawns a new thread that performs the rpc and terminates.

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 defined in the ODIN template;
- the reference variables which refer to 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 destructor that performs the termination phase.

5. RELATED WORKS

The approach of integrating C++ and DCE has been carried out also by [13], where transactional aspects, subtyping and notifications are not covered.

6. CONCLUSIONS

The development of the ODIN platform has confirmed that it is feasible to realise an object-oriented distributed platform with advanced capabilities built on top of commercially available platforms, by developing a limited amount of software, which does not introduce significant additional inefficiency to the underlying platforms.

ODIN represents a good approximation to the full support of TINA-compliant computational specifications and, thus, allows a faster implementation.

The choice of mapping the distributed processing model onto the facto standard and open platforms seems to be correct to achieve easy evolution and interoperability. Much work remains to do at the underlying platform level in order to extend the range of application to real-time service control. Instead of looking for proprietary implementation of the model, we would welcome more efficient implementation of the functionalities concerning distribution and concurrency provided by the underlying platforms (e.g., DCE).

Yet some open issues still remains, among them the most important are:

- introduction of persistent objects by integrating an OODB into the existing platform;

- alignment of the ODIN IDL to the OMG IDL [11].

7. REFERENCES

1. ISO/ODP, Basic Reference Model of ODP-part 3: Prescriptive Model, ISO/IEC JTC1/SC21 N8125 (June 1993).
2. P.G. Bosco, G. Martini, C. Moiso, A Distributed Object-Oriented Platform based on DCE and C++, in Proc. ICODP '93 (Sept. 1993)
3. N. Natarajan, F. Dupuy, N. Singer, Computational Modelling Concepts, TINA Consortium (Dec. 1993).
4. P.G. Bosco, G. Giandonato and C. Moiso, A distributed processing model for telecommunication services and operations software, in Proc. TINA'93 (Sept. 1993).
5. OSI/NM,Information Technology - OSI - Management Information Services - Structure of Management Information, ISO/IEC JTC1/SC21 N01165 (1991)
6. E. Moss, Nested Transactions, MIT Press (1985)
7. G. Weikum, Concepts and Applications of Multilevel Transactions and Open Nested Transactions, in Database Transaction Models for Advanced Applications (1992)
8. W. Weihl,Commutativity-Based Concurrency Control for Abstract Data Types, IEEE Trans. Comp. (Dec. 1988)
9. APM Limited, ANSAware 3.0 Implementation Manual, (Jan. 1991).
10. OMG, The common object request broker: architecture and specification, OMG Document Number 91.8.1 (Aug. 1991).
11. OSF, Introduction to OSF DCE,Prentice Hall (1992).
12. Transarc Corporation, ENCINA, Transarc Corporation (1991).
13. M.U. Mock, DCE++: Distributing C++-Objects using OSF DCE, in Proc. International DCE Workshop (Oct. 1993).