Cover Model: a Framework for Design and Execution of Distributed Applications

V.P. Ivannikov, K.V . Dyshlevoi, V.E. Kamensky, A.V. Klimov,

S.G. Manzheley, V.A. Omelchenko, L.B. Solovskaya, A.A. Vinokurov

Institute for System Programming, Russian Academy of Sciences

Bol.Kommunisticheskaya, 25, Moscow, Russia

phone: (7-095) 912-44-25

fax: (7-095) 912-15-24

email: dkv@ispras.ru

Abstract

Many problems of distributed object-oriented applications can be uniformly resolved in the frame of approach based on the concept of cover. The cover is defined as an environment which transparently controls all aspects of object's community life: creation, interaction etc. To enable transparency, an object-oriented application must obey a principle of late binding, a reference to server object being obtained by the client at run time from a system environment. To implement cover services, the technique of metaobject control is applied, which provides extensions of program's semantics without changing the program code, by means of attaching additional method calls to each application object invocation. A special language (TL) in which the user can incrementally define new metaservices is described and illustrated by numerous examples.

Keywords: object-oriented programming, distributed system, metaobject protocol, metaobject, interface object, cover, CORBA.

1 Introduction

The development of distributed object oriented applications implies many problems, concerning such aspects as object migration, coherence of contexts, adding new features to old objects etc. Most of these problems are targeted in our approach based on the concept of cover.

The life of objects must be organized. We define the *cover* as a receptacle for objects, which controls various aspects of their community life: creation, deletion, migration, interaction with other objects (method invocations), method parameter representation etc. Generally, an application may be distributed between many covers, each containing a group of semantically related objects.

An interaction between two objects is supported *transparently* by covers containing these objects. This implies the possibility of transparent extension of an application object with additional functionalities. The transparency means that the change of functionality is invisible for application program (that is for client and server application objects).

A *client* object invokes method on a *server* (target) object using an *object reference* which was implicitly supplied by the cover. To enable transparency, application program must follow a certain discipline: a reference to server object is obtained by the client at run time either directly from a system environment or from another such object. This principle will be referred to as *late binding*. In addition, the object's interface must be formally specified in an Interface Definition Language (IDL).

We do not fix a specific programming language, however we suppose it to be strictly typed, object-oriented and compilation-oriented. Therefore we need to distinguish two stages of object life cycle: *design stage* (including compilation) and *run-time stage* (method invocations). Covers support and manage the process of mutual conformation of objects at both stages. As covers are responsible for transparent coherent object interactions, they need some discipline of conforming themselves. We will distinguish between cover conformation at design stage (static cover conformation) and that at run time (adaptive dynamic cover conformation).

For flexible cover conformation we use the technique of *metaobject control* [KRB91] which is usually applied either at design stage or at run-time stage.

In the first case [Chi96] we get a high-performance but rigid code, i.e. we have not opportunity to change metacontrol at run time. In the second case, we pay, for the ability of unrestricted dynamic change of object method invocation, with a significant slow-down (at least for one decimal order of magnitude) because of interpretation [ZC95, ZC96].

Our approach to metaobject control [IZKN96, IDZ97] is a compromise between these alternatives. The main idea is the following. At the design stage we define different variants of possible metacontrol for object interaction (the absence of metacontrol is also possible). So we get several variants of high-performance code. And the final selection of a variant is made at run time.

2 Related works

The cover approach is based on some basic ideas: orthogonal decomposition of system, metaobject control performed by means of well-developed technique of mediator objects, conformation of distributed system's components called covers.

Orthogonal decomposition of systems was considered in the context of aspect-oriented programming project [KASP]. Traditional approach of module decomposition was criticized as it does not reflect a difference between such system features (aspects) as multi-access, communication, debugging, etc. Authors of the project suppose that it is tangling-of-aspects that is the basic reason of complexity of many programming systems. The aspect-oriented programming allows a programmer to describe different aspects of the system in different ways (e.g. in different programming languages) and point out relations between these aspects. Then a special tool Aspect Weaver helps to generate final execution form in some programming language according to these descriptions.

In operating system Tigger [ZC96] orthogonal decomposition is also performed. Any object method call can be intercepted before and/or after the method execution. Special system component (Piglet Core) refers such interceptions to some metaobjects which are grouped according to orthogonal functionality (metaregions).

Metaobject control and reflection are well-known technique used to design flexible dynamic adaptive programming system in object-oriented programming ([KRB91], [MJD96]). Some popular programming system are based on this technique ([FDM94], [DF94], [NH96]). Sometimes metaobjects play the role of mediators between communicating objects [GC96]. This project assumes metaobjects to be specialized according to reification kinds: object methods invocation, access to object state, access to program code at run time, etc.

Transformation of object method invocation at run time needs the usage of some mediators between objects. Sometimes, such mediators are implemented as objects themselves according to object-oriented technique. ORBIX [IONA96] provides objects mediators of several kinds. First of all, we can name filters. Filters allow to add some before- and after-functionality to particular object, as well as to all objects working via broker (ORB). In fact, filters perform method invocation trapping giving possibility to add additional parameters to method call on the client side and to work on these parameters on the server side. But conformation between client and server is not supported and must be provided by application itself. Another mediator is smart proxy. This object is implemented in some programming language and replaces the target object. In principle, smart proxy can invoke methods on other objects as well as on the target object. CORBA Security Service [COSS96] introduces technique of intercepter similar to filter mechanism. Such filters can add some client information to the request passed by broker and then check it on the server side.

OMG project on multiple interfacing introduces another kind of object mediator, an interface [MI97]. It basically serves to provide several coherent interfaces to an object and allow client to access these interfaces. Interface transforms nothing and performs parameter passing only. That is, interface generalizes the notion of object reference.

Cover approach is an attempt to compile together advantages of all these techniques.

3 Cover features

3.1 Run time cover features

At run time the cover serves as an environment, in which objects exist and communicate with each other. The cover controls the life cycle of objects, that is creation and deletion of objects. Also, it organizes the communication with the environment, including objects from the same cover and those of other covers. The cover controls all general aspects of all objects contained in. In particular, it provides a discipline of getting access to objects.

The cover performs its control over application objects by means of a special kind of objects called *metaobjects*. All cover’s metaobjects form the *metacontext* of the cover. As metaobjects are objects themselves, they can be controlled by other metaobjects in turn.

We suppose the metacontext to be divided into functionally independent (*orthogonal*) collections of metaobjects, called *metaservices*. Each of these metaservices provides one or another kind of functionality, e.g. multiaccess, persistency, statistics, debugging etc.

The orthogonality of metaservices implies the following features:

• ***transparency***: binding application objects to various functional elements of metacontext is transparent for application object as well as for the other metaservices;

• ***independency***: different *metaservices* are developed independently of each other;

• ***extensibility***: adding new metaservices (*metaupgrade*) does not require any changes of existing metaservices.

Note, that the complete orthogonality is generally an ideal rather than true objective. For example, in some algorithms the supports for multiaccess and persistence are tightly integrated. In such cases we need to have a combined metaservice multiaccess+persistence, instead of two separate ones.

Typically, metaservices are independent of server object implementation. In other words, application objects can be considered as “black boxes”. This assumption is feasible for many kinds of metaobject control (e.g. synchronization, statistics).

However some metaservices may require more close access to the properties of the target object. Often, some public methods of application object are sufficient for this purpose. However, when application object does not provide the required methods, a metaservice (e.g. persistency) might need a direct access to the application object’s encapsulated state.

The basic mechanism of object interaction inside the cover as well as between covers is provided by the so called *communication kernel*, which is a part of each cover. Depending on mutual location of objects taking part in an interaction (local or remote) the kernel provides an appropriate communication scheme. Due to this mechanism, two objects interact more efficiently when they are in the same address space. Thus, it is the cover kernel which provides location transparency of the objects.

To support conformation between objects and external environment at run time, each cover provides all needed information about included objects and some mechanism of searching objects by name. That is, the cover performs the functions of Naming Service. Name resolution is realized not only in the scope of current cover. The covers can be nested and/or joined in a federation in order to provide common name space for search [ANSA93].

3.2 Design stage cover features

For design purposes three languages are used: implementation language, interface specification language and language for description of metaobject bindings. Since we would like to have reliable, high-performance code, the implementation language must be strictly typed object-oriented language, e.g. C++. CORBA Interface Definition Language (IDL) is a good candidate for specification language as it guarantees the fulfillment of the requirement of compatibility with CORBA [CORBA95].

Also we need a language in which to describe various kinds of bindings of target objects to metacontext, that is the body of IO. These bindings can be expressed in the form of generic schemes (templates) specifying interaction with the cover elements. The language to specify such schemes must meet the following requirements:

• ***Strict typing***. It must be a strictly typed language.

• ***Generic types***. It should allow for parametric description (using generic forms) and a stepwise refinement of binding schemes until the type of the IO is completely defined.

• ***Orthogonality***. It provides opportunity for independent description of binding with orthogonal metaservices.

• ***Composability***. It allows for various metaservices to be composed in a simple way.

• ***Clustering***. It serves to describe variants of bindings.

Current name of the language is TL (Template Language). Compiler of IO is fed with both TL and IDL texts as input, and produces text of IO in the implementation language as output. At the design stage, to enable static cover conformation we use descriptions of own local objects in all three languages, and descriptions of external covers and of objects contained in them in the TL and IDL languages. The process of conformation of a cover A with some external cover B is performed by means of concretization of TL declarations of this external cover B. Specifically, the generic templates describing the cover B are instantiated into definitions of IO of cover A, by means of substitution as actual parameters of various elements of cover A (metaobjects, metaobject method calls, IO state elements, etc.). Thus, we make metacontext of cover A coherent (statically) with that of cover B.

At the design stage the cover is used to describe groups of objects of different types related to specific tasks. As such it serves as a repository, which contains information needed to specify included objects, i. e. meta-information. For example, the cover provides information about object interfaces and parameter types of their methods. Besides, the cover contains information needed for dynamical conformation of the covers at run time.

All information provided by the cover at the design stage is used for the design of a coherent object space inside the cover, as well as for the design of other covers, whose objects could interact with this cover and its elements.

4 Cover implementation by means of Interface Object technique

The cover model implementation is based on the technique of metaobject control, defined as follows. We extend the semantics of application objects not due to changes in the code of client and server objects but rather by a special execution of invocation. Additional method calls are added to the client call of a server object transparently for the user. These additional methods are implemented by metaobjects which control the behavior of appropriate application objects.

The metacontrolled application object is spoken about as bound to the metaobject environment (*metaenvironment*) of the cover. Using covers to extend application semantics is called *metaextension*. The cover approach supports also the evolution of the system, providing a smooth way of adding new metaservices to the cover’s context (*metaupgrading*).

Our approach differs from that based on reflection and the meta-level interpreter [MMAY95] which involves the major problem of efficiency. What we propose may be considered as direct and efficient implementation of reflection for method invocation.The main idea is to use a special kind of metaobject called Interface Object (IO) as a mediator between a client and a server objects. IO usually has the same application interface as that of the server object, and some additional control interface to enable run-time change of metaobject control. It is IO, which binds object to metacontext and organizes additional metaobject calls around the application call to the server.

To enable IOs to be transparently inserted, the interacting objects should not be bound statically. Rather, application objects should find each other with the help of some system environment, which is not part of the application. It is this system environment that forces application objects to communicate via IO if necessary. Once a single object is represented by an IO, all invocations over it are controlled by the cover. In particular, when another object reference is returned as a result of an invocation, the respective metacontrol should return an IO instead of this object reference.

Often, IO can be split into two objects: IO on the client side (client IO) and IO on the server side (server IO). Client and server IO can be independent of each other to some degree. Conformation of these IOs is supported by appropriate covers.

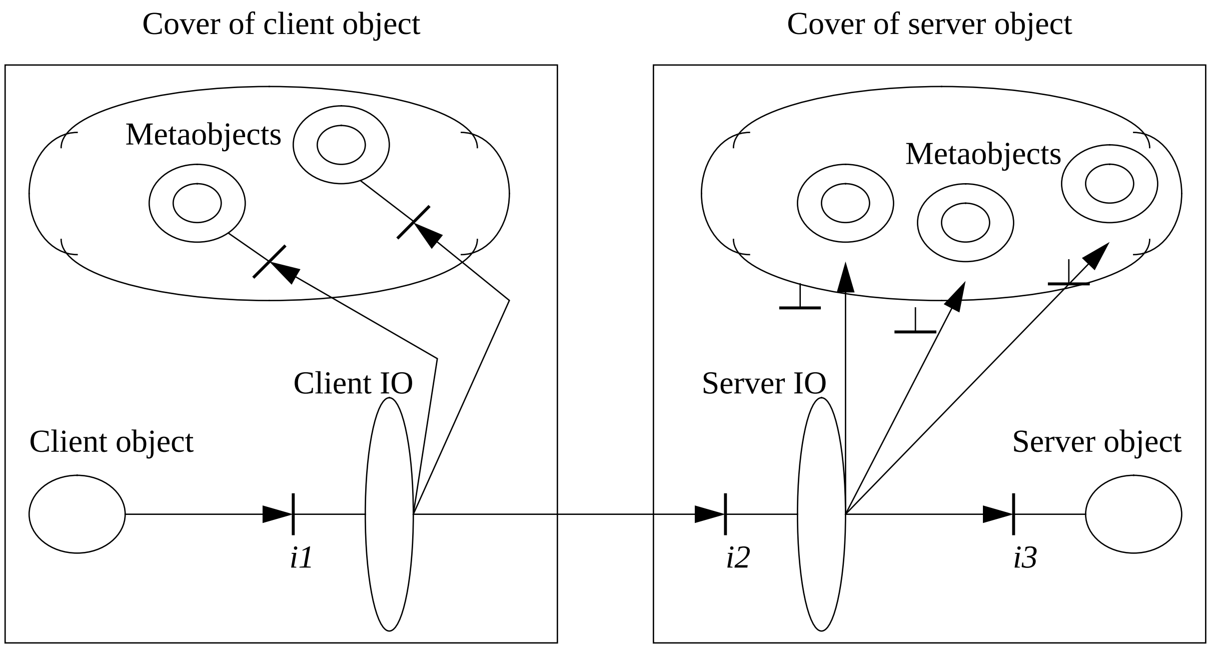

Figure 1. Metacontrol organization scheme.

Figure 1 illustrates the scheme of objects interaction by means of client and server IO. The agreement on designations is the following. Application objects are defined by circles, IO by ellipses, metaobject by double circles.

The real interface the client uses is the interface of a client IO (*i1*). Client IO calls methods of server IO (*i2*). Finally, server IO actually invokes methods of target object interface (*i3*). Note, that here interface *i2* need not be the same as *i3*, whereas *i1* is expected to be the same as (a subset of) *i3*.

Also, either client or server IO can be used alone, thus making a binding to the metaenvironment of the respective cover only. When neither client nor server IO is used, metacontrol is not performed.

In case of remote invocation, since the *marshalling* of parameters according to some protocol is necessary, the client and server IOs organize communication via the cover kernels. Each cover kernel performs all necessary data transformation and makes actual data transmission to another cover kernel. Since an interface object and the corresponding application object belong to one and the same cover, the communication between them does not require any data transformation and does not depend on mutual location of application objects. Thus, we say that a pair of IOs used on both ends of remote interaction provides the location transparency of application objects.

5 Means for specification of binding object to cover metaenvironment

Recall our agreement that interfaces of all objects (including IO and metaobjects) are specified in CORBA IDL language. We do not fix here the concrete programming language which is only assumed to be strictly typed and, in general, object-oriented.

Another specification language, named TL, serves to describe variants of metacontrol schemes for client and server metaenvironment. Metacontrol descriptions in TL do not depend on concrete application and programming environment. In general, metacontrol scheme means the sequence of metaobject invocations, which are encapsulated in IO specification. Each IO has a set of parameters and statically defined metacontrol schemes, which can be customized at run time.

To illustrate the features of TL language consider the following example.

5.1 Metacontrol specification example

Two objects O1 and O2 exchange information represented by strings. Object O1 calls on object O2 which implements some interface T with the method:

```
exchange_info (IN string arg_info, OUT string res_info);
```

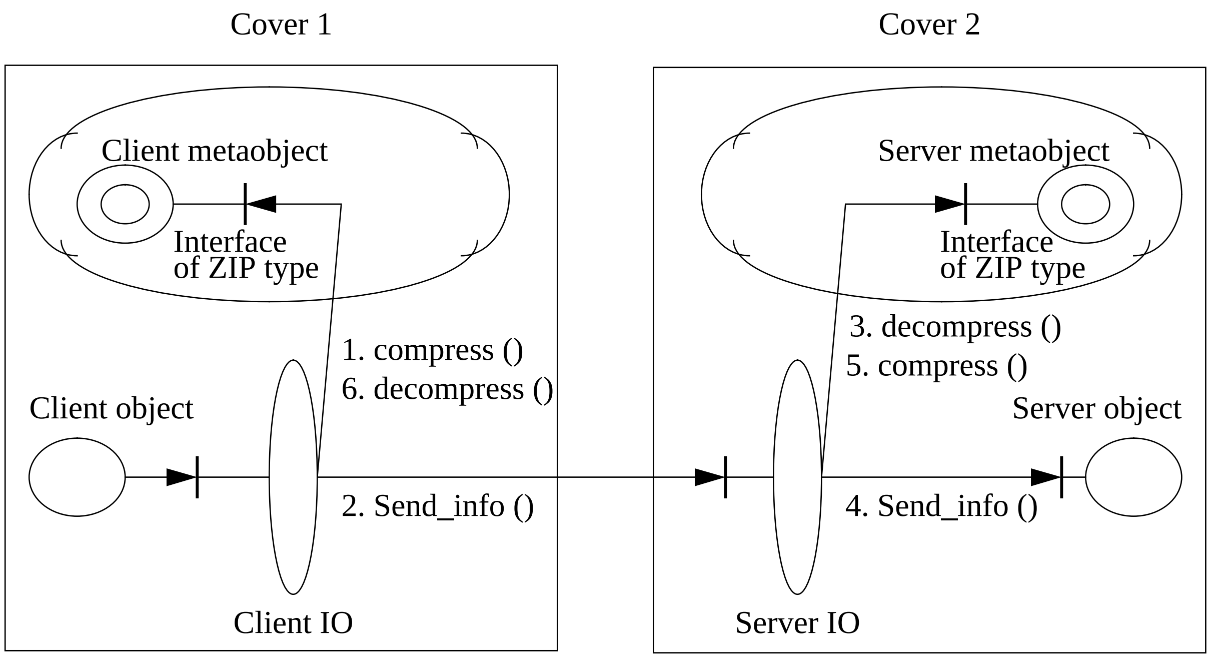

Figure 2. Metacontrol organization example.

In case of remote invocation, the subject of metacontrol can be data compression /decompression used to reduce the size of passed data. To organize such metacontrol we need two IOs (client and server IO), which compress the data to be sent to, and decompress the data received from the net.

There are metaobjects implementing some interface ZIP with compress and decompress methods:

```
void compress (IN string info, OUT string compressed_info);
void decompress (IN string compressed_info, OUT string info);
```

Covers C1 and C2, which contain respectively the objects O1 and O2, create each its own metaobject of type ZIP and bind it to the corresponding IO.

Metacontrol for operation *exchange_info* of client and server IO is described in TL as follows:

```
T {
    // common declarations for client and server IOs
    OBJECT Zip_metaobject : ZIP;
    SERVER IS {
        exchange_info (compressed_arg, compressed_res) USE {
            variable arg_info : string;
            variable res_info : string;
            Zip_metaobject . decompress (compressed_arg, arg_info);
```

```
                    // target object’s method call:
                    send_info (arg_info , res_info);
                    Zip_metaobject . compress (res_info, compressed_result);
                };
                // server  IO’s metacontrol description for other methods
                ...
            };
            CLIENT IS {
                exchange_info (arg_info, res_info) USE {
                    variable compressed_arg : string;
                    variable compressed_res : string;
                    Zip_metaobject . compress (arg_info , compressed_arg);
                    // server IO’s method call:
                    send_info (compressed_arg, compressed_res);
                    Zip_metaobject . decompress (compressed_res, res_info);
                };
                // client IO’s metacontrol description for other methods
                ...
            };
        };
```

Object reference Zip_metaobject of type ZIP is initialized by the cover in both client and server IO. For both IOs, implementation of method *exchange_info* contains two invocations on metaobject and an invocation on target object itself. Figure. 2 illustrates all invocations numbered in the order of execution. Note, that for the client IO the target object is the server IO, whereas for the server IO the target object is O2. To manage the data transfer, some local temporary variables turned out to be necessary; these are compressed_arg and compressed_res in the client IO and arg_info and res_info in the server IO.

Also another variant of metacontrol, when data are transmitted without changing, is possible. Alternatives of metacontrol schemes allow client and server to consistently choose the variant of conforming metacontrol at run time. The metacontrol switching mechanism is implemented by means of the cluster technique (described in detail in section 4.7).

5.2 Basic means for metacontrol description

Each scheme of metacontrol has a block structure with a set of local variables encapsulated in each block, the body of a block being a sequence of calls interconnected by parameters. In TL blocks can be nested and can contain local variables declaration section. Standard scope rules are used for variables encapsulated in blocks. Also, some blocks may have labels in order to allow transitions between them. Block declaration looks as follows:

```
[Block name] {
        variables declaration section;
        invocations; blocks declarations;
}
```

Each invocation consists of a reference to an object, a method name, and a list of parameters. The object reference can be omitted in the IO’s target object invocation only.

5.3 Metaobject environment specification for applications

Each metaobject used in IO must be declared in TL as in the following line of the example above

```
OBJECT Zip_metaobject : ZIP;
```

It declares a data member and the respective attribute (in the sense of IDL) of the IO’s interface. This attribute is accessible for cover, not for the client. More exactly, it is accessible for some cover’s metaservice which manages conformation of IO and metaenvironment. Thus IO has two different interfaces: a visible for clients application interface and a hidden from clients *metainterface* being used by cover services for initialization and later changes of IO state.

Figure 3 shows both IO interfaces. Dotted arrows designate full manageability (including the possibility of modification), while solid lines designate only usage of pointed elements. Changing appropriate object reference the cover can dynamically switch one metacontrol scheme to another.

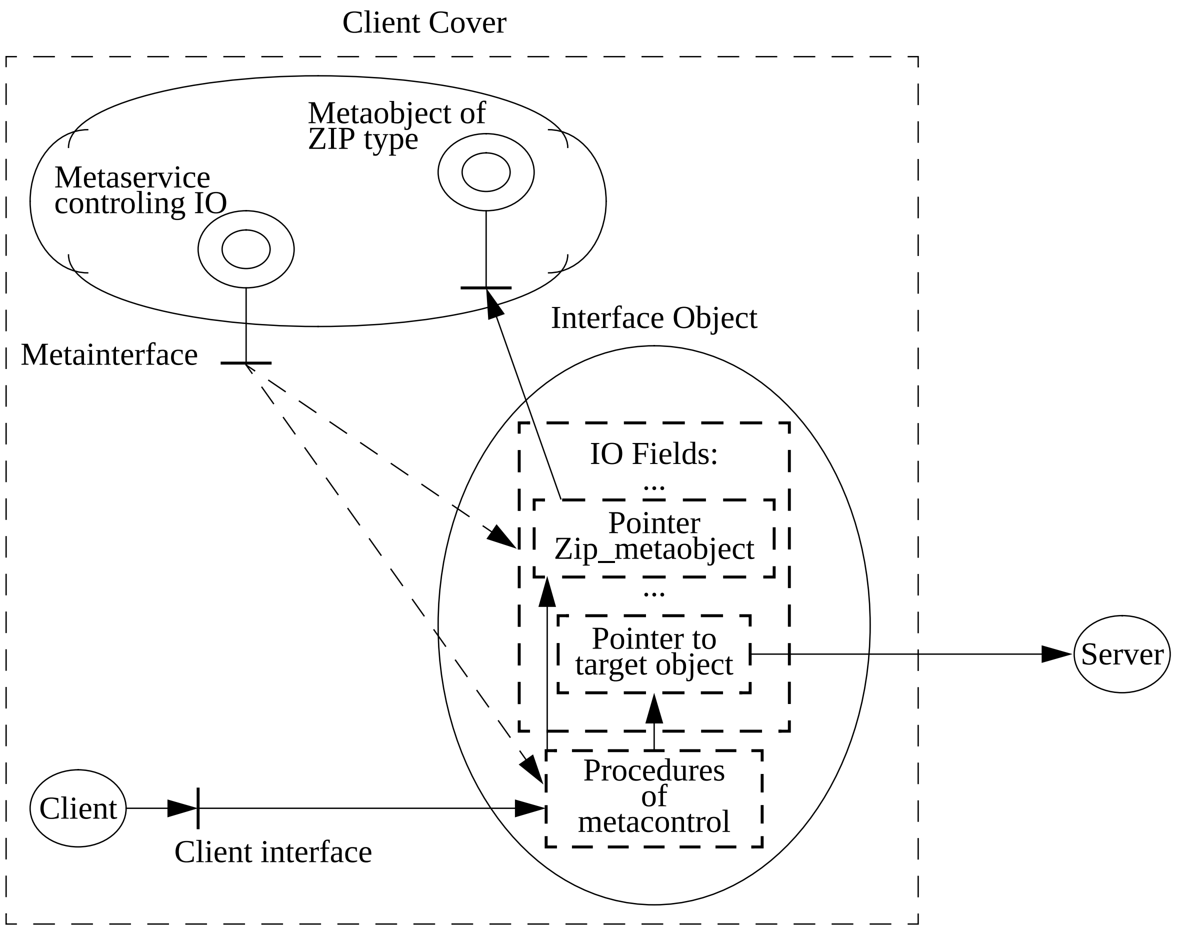

Figure 3. Scheme of working with interface object.

5.4 Templates libraries

Semantics of a cover's service determines typical sequences of service method invocations. In this way, certain sequences of method calls can be present as generic templates (patterns) and form templates libraries for one or another type of services.

For example, access controlling service (concurrency service) having two operations (lock that acquires object access and unlock that releases object) can be declared as a following template:

```
LockPattern ( rw_flag ) < concurrency_objref > { ControlledOperations } IS {
        concurrency_objref . lock ( rw_flag );
        DO ControlledOperations;
        concurrency_objref . unlock ;
}
```

In this template description LockPattern is a template name. Parameters of template are rw_flag (lock operation parameter), concurrency_objref (the name of access control service) and

ControlledOperations (controlled operations).The last parameters kind allows one to use template for several methods of target object having identical metacontrol.

An example illustrating the usage of concurrency service template follows. We assume IO to have an appropriate attribute declaration:

```
// reference to Concurrency metaservice
OBJECT Concur_service : Concurrency;
```

Method *some_operation (in SomeType arg)* can be metacontrolled in IO with the help of concurrency service template:

```
some_operation (arg) USE {
        LockPattern (WRITE) < Concur_service > { OPERATION }
}
```

Keyword OPERATION used in above specification means target object's method invocation (here it is *some_operation* method) with the given IO parameters (in this example it is *arg* parameter). This keyword allows to define identical metacontrol for group of methods in a single specification:

```
method_1 (...), method_2 (...), ..., method_n (...) USE {
        ...
        OPERATION;
        ...
}
```

5.5 Using templates compositions

The usage of templates and template libraries allows one to abstract from details of method invocation in a given service and think in terms of refinements of template compositions. The assumption of orthogonality of services makes the concept of template composition very natural.

To illustrate the possibility of template composition we continue example from the previous section. *Some_operation* method, which is already wrapped with synchronization locks, can be metacontrolled by statistics service too. For this purpose IO must have another attribute referencing to statistics service of type Statistic:

```
OBJECT stat_obj : Statistic;
```

This service has *begin_time* and *finish_time* methods to store start and finish time of some activity.

```
StatisticPattern <statistic_objref> { ControlledOperations } IS {
        {
                statistic_objref . begin_time ;
                DO ControlledOperations ;
                statistic_objref . finish_time ;
        }
}
```

A composition of patterns LockPattern and StatisticPattern declared above may look as follows:

```
some_operation (arg) USE {
        LockPattern (WRITE) < Concur_service > {
                StatisticPattern < stat_obj > { OPERATION }
        }
}
```

Nesting in this order provides lock-unlock service for both controlled operation and statistics service calls. If it were desirable to protect just server object call, the nesting should be inverse:

```
some_operation (arg) USE {
        StatisticPattern < stat_obj > {
                LockPattern (WRITE) < Concur_service > { OPERATION }
        }
}
```

5.6 Using clusters

To switch statically defined metacontrol schemes at run time the mechanism of cluster is used.

A specific data member (IDL attribute) of enumeration type is added to the IO. Changing value of this attribute the cover can change the metacontrol scheme. We can say that a set of clusters for IO is provided. Cluster's identifiers serve as values of enumeration type of the respective IO attribute.

Example of statistics service illustrates the working with a cluster set. A set of clusters StatisticClusterSet having two elements (clusters) MakeStatistic and NoStatistic is declared. Alternatives of metacontrol schemes are declared in StatisticCasePattern template. These declarations are placed into library of statistics service:

```
TEMPLATES LIBRARY StatisticLib {
    StatisticPattern <statistic_objref> { ControlledOperations } IS {
        statistic_objref . begin_time ;
        DO ControlledOperations ;
        statistic_objref . finish_time ;
    }
    CLUSTER SET StatisticClusterSet {
        DEFAULT MakeStatistic, NoStatistic };
    StatisticCasePattern <service_ref> { ControlledOperations } IS {
        MakeStatistic :
            StatisticPattern < service_ref> { ControlledOperations };
        NoStatistic : DO ControlledOperations;
    }
}
```

Now, to use this library's contents in metacontrol definitions one should refer to it explicitly:

```
USE TEMPLATES LIBRARY StatisticLib;
```

Such library referencing leads not only to a possibility of using its templates, but to automatic addition of StatisticClusterSet attribute to IO. The usage of template with clusters (StatisticCasePattern) is the same as that without clusters.

In general, any IO can use any number of libraries with clusters definitions at the same time. That is, IO may have any number of attributes varying independently of each other. These attributes make it possible to manage orthogonal metaservices independently.

5.7 IO sequential refinement technique

In the previous example Statistics service was separately described first and then Concurrency service was added. Such step-wise refinement seems very natural. The possibility of step-by-step metacontrol definition is included in TL language.

IO definition can be developed as a sequence of refinements. Each of such refinement has its own identifier (name) and is a generic description for some part of metacontrol. So, description of the working with statistics service (example from section 4.6) can be considered as the first refinement.

(Interf is an interface having *some_operation* method; this interface is used in the context of one cover, therefore only server IO is defined):

```
Interf {                    // First refinement of IO
    USE TEMPLATES LIBRARY StatisticLib;
    OBJECT stat_obj : Statistic;
    SERVER FirstVersion < other_services {} >
    IS {
        some_operation (arg) USE {
            StatisticCasePattern < stat_obj > {
                other_services { OPERATION } }
        }
    }
}
```

This specification of IO contains only binding of Statistic metaservice. Parameter other_services of this refinement named FirstVersion leaves the possibility to extend *some_operation* method call in the further refinements. Next refinement uses descriptions from the first one (it is referred by keyword REFINES) and adds working with metaservice Concurrency through parameter substitution:

```
Interf {                    // Second refinement of IO
    USE TEMPLATES LIBRARY ConcurrencyLib;
    OBJECT concur_service : Concurrency;
    SERVER SecondVersion
        REFINES FirstVersion < ConcurCasePattern (WRITE) <concur_service> >;
}
```

Note, that this refinement does not contain parameters, but still can be refined. For example, in the next refinement one can use another metaservice for client IO description:

```
Interf {                    // Third version of metacontrol description
    CLIENT ThirdClientVersion { some_service_pattern {} }
        REFINES SecondtVersion
    IS {
        some_operation (arg) USE {
            some_service_pattern {
                SecondVersion::some_operation (arg) }  };
    };
};
```

Obviously, such separated IO definition is not just convenient for IO development representation, it also simplifies possible redesign of metacontrol, because usages of different metaservices are localized in different refinements. Besides, IO refinement process makes it possible to separate the development of client IO at the design stage in server and client covers in quite natural way. At the server design stage, only the metacontrol from the server' metacontext can be fully defined for client IO. Server cover defines only those metaservices in client IO that are necessary for conforming interaction with server IO. In general, the client IO metacontrol is refined in the client design stage cover. Client cover can both refine the usage of metaobjects and add new orthogonal metaservices.

6 Conformation with CORBA technology.

This section is intended to discuss how the above mechanism of covers and the usage of metacontrol can be arranged in CORBA, the widely-known architecture for communicating objects in distributed systems. Note that the interface definition language IDL we use is borrowed from CORBA.

CORBA architecture uses mediator objects as object references. On the client side this mediator is called stub, on the server side skeleton. Neither stub nor skeleton are parts of the application (in fact, they are parts of a system environment).Their only purpose is to provide location transparency.

Note that a CORBA conforming application suits well to metacontrol owing to the mechanism of late binding CORBA relies on. The question arises, how does IO, which is also a mediator between client and server, relate with stub and skeleton.

On the server side, both the server object itself and the server IOs are created and controlled by the cover containing them. Thus, following CORBA, the cover should create a skeleton and bind it with the server object, so that each method call be passed via the skeleton as a mediator. So, both the server IO and the skeleton play similar roles in operating with target object. Hence, on the server side, these two mediators can be united in the sense that the functionality of IO can be added to the skeleton.

On the client side, the relation between stub and IO is more complicated. Stub, as opposed to skeleton, cannot be created by the cover. It is created by internal ORB's mechanisms when the client receives an object reference. If stub and IO were unified in a single mediator, the cover would lose the ability to control the life time of the IO and to select IO needed by a given client. Also, the stub would be too complicated in case it is used just as an object reference to be passed to another object.

To associate IO with stub different algorithms must be used. For example, when the client invokes on the stub for the first time, the stub requests the client cover for an IO. Depending on the type of the stub, the cover selects a client IO needed by a particular client. After the stub receives a reference to the associated client IO from the cover, all method calls including the first one are passed to the IO.

As for the standard CORBA services, the metacontrol technology allows to use them along with any other services either as cover metaservices or as parts of user applications. We do not regulate anyway their usage which remains up to the designer an object system in the frame of the cover model.

7 Conclusion

We defined the notion of cover which serves as a framework for both building and running object-oriented applications. A lot of hard problems of distributed systems can be resolved smoothly within our approach: conformation of distributed contexts, location transparency, object migration etc.

The cover performs the total control over the object method invocations in application program provided that the application obeys the principle of late binding. In particular, we have seen, that this principle is valid for CORBA compliant applications. In a sense, our approach may be considered as a generalization of CORBA model of object interaction.

8 References

[ANSA93] Rob van der Linden. The ANSA Naming Model. Architecture report APM.1003.01, Poseidon House, Cambridge, 15 July 1993

[Chi96] Shigeru Chiba. A Metaobject Protocol for C++. OOPSLA'95, 1995, pp. 285-299.

[CORBA95] The Common Object Request Broker: Architecture and Specification. Revision 2.0, July 1995.

[COSS96] CORBA services: Common Object Services Specification, OMG Document 95-3-31, 1995. Updated: November 22, 1996.

[DF94] Scott Danforth, Ira R. Forman. Reflections on Metaclass Programming in SOM.

OOPSLA 94- 10/94 Portland, Oregon USA

[FDM94] Ira R. Forman, Scott Danforth, Hari Madduri. Composition of Before/After Metaclasses in SOM. OOPSLA 94- 10/94 Portland, Oregon USA

[GC96] B.Gowing, V.Cahill. Metaobject protocols for C++: the Iguana approach. Proceedings of Reflection 96 Conference. 1996.

[IDZ97] Ivannikov V., Dyshlevoi K., Zadorozhny V. Specification of metaupgrade for efficient metaobject control. Programmirovanie, N4, 1997.

[IONA96] The Orbix Architecture. IONA Technologies. November 1996. Http://www-usa.iona.com/Orbix/arch

[IZKN96] Ivannikov V., Zadorozhny V., Kossmann R., Novikov B. Efficient Metaobject Control Using Mediators. Proc. Perspectives of System Informatics. Novosibirsk, Russia. LNCS 1181, pages 310-330. June 1996.

[KASP] Gregor Kiczales et.al. Aspect-Oriented Programming. A Pos. Paper from Xerox Parc.

[KRB91] Kiczales G., des Rivieres J., Bobrow D. The Art of the Metaobject Protocol. MIT Press, 1991.

[MI97] Multiple Interfaces and Composition. OMG Doc. orbos/97-02-06. February 1997.

[MJD96] Malenfant J., Jacques M., Demers F.-N. A Tutorial on Behavioral Reflection and its Implementation. Proc. of Reflection’96 Conf., San Francisco, USA, April, 1996.

[MMAY95] Masuhara H., Matsuoka S., Asai K., Yonezava A. Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. OOPSLA'95, 1995, pp. 300-315.

[NH96] Farshad Nayeri, Ben Hurwitz. Generalizing Dispatching in a Distributed Object Systems. ECOOP'96.

[ZC95] Chris Zimmermann and Vinny Cahill. How to Structure Your Regional Meta: A New Approach to Organizing the Metalevel. In Proceedings of META'95, a workshop held at the European Conference of Object-Oriented Programming, 1995.

[ZC96] Chris Zimmermann and Vinny Cahill. It's Your Choice - On the Design and Implementation of a Flexible Metalevel Architecture. Proc. of the Int. Conf. on Configurable Distributed Systems, IEEE, Annapolis, Maryland, May, 1996.