Distributed applications design and run time support in Cover Model

V.P. Ivannikov, K.V. Dyshlevoi, V.E. Kamensky, A.V. Klimov, S.G. Manzheley, V.A. Omelchenko, L.B. Solovskaya, A.A. Vinokurov

1. Introduction

Conceptually we consider the set of all objects of the distributed global space as covered by subsets of a special kind - covers. Each cover contains a community of objects coupled together semantically. Any cover is a special object itself. Covers can be nested and can be unified into federations.

An interaction between two remote objects is supported transparently by covers containing these objects. The transparency means that objects do not see covers containing them. An object invokes methods on another object independently of:

- object locations (local or remote);
- features of interacting covers (object model supported by concrete cover; implementation of multiaccess to cover's objects; methods used for security support, etc.);
- communication ways between covers (IIOP CORBA 2.0 [CORBA95] or another specific protocol).

Without the loss of generality, we suggest the implementation language to be strictly typed and object-oriented. Therefore we ought to distinguish two stages of object life: design stage and run-time stage (object method invocations). Covers support both stages and manage the process of mutual conformation of objects on both stages. In other words, covers form both a framework for object design and an environment for run time object invocations.

Now that covers are responsible for transparent coherent object interactions, they need some discipline of conformation themselves. In principle, we could represent a homogeneous world of covers, according to some unique standard, for example, CORBA [CORBA95] (CORBA-covers). However we would like to suggest richer world of cover models, possessing not only various features, but also features for description of these features (metafeatures). Since we deal with typed languages, we need to distinguish between conformation at design stage (static cover conformation) and conformation at run time (adaptive dynamic cover conformation).

For flexible cover conformation it is natural to use the technique of metaobject control [KRB91]. However, usually, metaobject control is used either at design stage or at run-time stage.

In the first case [Chi96] we get high-performance code, but a monolithic one, i.e. we have not opportunity to change metacontrol at run time in a systematic way. But often distributed applications require some changes in conformation schemes during run time, for example, because of object migration (object is moved to another cover with other features) or because of changes of the cover's features (e.g. changes of security requirement). In the second case, the pay-off for the exceptional flexibility of unrestricted dynamic transformations of object method invocation is a substantial 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 prepare different variants of possible metacontrol for object interaction (the absence of metacontrol is also possible). And thus we get several variants of high-performance code. At run time stage one or another variant is used. Moreover, the current variant can be replaced at run time by another one, which is particularly useful if dynamic conformation of covers is required.

Let us consider the internal structure of cover. As it was already said, it contains some community of application objects, coupled together semantically. Also, in a cover, there is a metacontext, containing metaobjects. Metaobject can control behavior of application objects extending them with additional functionality. For example, metaobject can support multiaccess to application objects, or provide their persistency. According to the reflection principle, metaobject can be controlled by other metaobjects in turn.

Metaobjects, as well as application objects, follow the common rules of visibility within nested covers and covers federations (see section 6).

We suppose the metacontext to be orthogonally decomposed, that is 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 assumption of orthogonality is extremely important. It provides the following capabilities:

- to bind application objects to various functional elements of metacontext in a transparent way;
- to develop different functionalities (metaservices) independently;
- to add new metaservices without any changes of existing metaservices (metaupgrade).

It is important to note, that the complete orthogonalization is generally impossible. For example, there are some algorithms, in which multiaccess and persistency support are tightly integrated. In such cases we will act in a pragmatic way, introducing new combined metaservice - multiaccess-persistency.

To bind application object to metaobjects, we shall use a special kind of metaobject called interface object (IO). In terms of the client-server model, a client object may invoke a method on a server object either directly (without binding to metacontext), or via an appropriate IO, which acts as a mediator. Methods of remote object are always invoked via IO. On the one hand, IO inherits interface of target (server) object, that is IO provides all signatures of target object methods, and on the other hand, IO as metaobject has additional methods for making control over invocation execution. IO can be considered as a communication object, which performs invocations on metaobjects methods as pre- and post-functionality within each target object method invocation. To organize such pre- and post-functionality the orthogonality of metacontext components is used extensively.

It is important to note, that to maintain the transparency of using IO, some mechanisms of late binding must be used. That is, there is no static binding of objects, which were worked out at the design stage and could not be changed at run time. Rather, application objects communicate with each other by means of some system environment, which is not part of the application. It is this system environment that allows application objects to communicate transparently via IO if it is needed.

Using IOs can provide different kinds of relationship between objects: “one-to-one” (client and server object have one IO for their exclusive use), “many-to-one” (*all* client objects invoke methods on one server via one IO), “one-to many” (client object invokes methods on different server objects via one IO), “many-to-many” (several client objects interact with several server objects via one IO in multicasting mode).

Remind that binding to metacontext is transparent for both client and server objects. Metacontrol organization does not concern implementation of application (neither client nor server). In the sense, application objects can be considered as “black boxes”. This scheme is convenient for many kinds of metaobject control (e.g. synchronization, statistics).

However some kinds of metacontrol may require more close access to properties of target object. Often, some public methods of application object are sufficient for this purpose. However, when application object does not support the required methods, a metaservice (e.g. persistency) might need a direct access to the application object’s encapsulated state. This access can be implemented in different ways for different programming languages. For example, in C++ language, in order to have access to object state, metaobjects can be implemented as “friends” of target object [Str94].

It should be noted, that IO can be split into two objects: IO on the client side and IO on the server side (as stub and skeleton in CORBA model). IO of client and that of server can be independent of each other to some degree. Conformation of these IOs is supported by appropriate covers. As an example of such IO conformation one can take a transparent compression-decompression of data transmitted between interacting objects. At run time by means of both covers, one can make simultaneously some coherent changes in client and server IO bindings with metaobjects performing the required transformations. This additional data transformation may be switched on or off by the covers by means of binding/unbinding IOs to/from metaobjects.

Here the orthogonalization also simplifies the problem of division of dependent and independent components of metaobject control, as well as their changes at run time.

Let us consider briefly some aspects of cover model implementation and specific cover features at design stage. 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. We do not consider in this document how we could obtain IDL specifications of target objects. They can be either written by hand or obtained by a compiler from the implementation language to IDL.

Also we need a language in which to describe various kinds of bindings of target objects to metacontext (that is the contents of IO). This language must meet the following requirements:

1. Strict typing. It must be a strictly typed language.
2. 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.
3. Orthogonality. It provides opportunity for independent description of binding with orthogonal (independent) metaservices in the process of refinement.
4. 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 implementation language as output. To support the design stage of the cover we need 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 with that of cover B.

So, the cover approach described above 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 - covers.

Orthogonal decomposition of system is described in the context of project of aspect-oriented programming [KASP]. Traditional approach of module decomposition is criticized in the sense that it does not reflect difference between system element's features (aspects) such 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 existing programming systems. Aspect-oriented programming allows programmer to describe different aspects of 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 or/and after 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 invocations trapping, giving possibility to add additional parameters to method call on the client side and 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 advantages of these techniques. It is described in detail in the following sections.

2. Notion of cover

To construct and manage logically coherent objects in the distributed global space the notion of Cover is introduced. The cover is a separate object itself. The cover serves to arrange objects at both the design stage and the run-time stage.

The main idea of the approach is that the cover model allows for the extension of application semantics with additional semantics without changing the application. In other words, application objects can be included in cover contexts in the transparent manner. So, the cover can manage objects that were designed knowing nothing about this model.

At the design stage, the cover is used to group objects of different types related to specific tasks. It can be understood 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. It is important to notice, we assume this information to be strictly typed. That is, this information should be enough to organize invocations on the cover and its elements by means of any strictly typed programming language (C, C++, Pascal, etc.).

Also, at the design stage the cover allows to specify typical schemes of interaction with the cover elements. These schemes reflect the semantics of cover contents. They are specified by means of generic templates, which fix, according to the standard rules, the mechanisms for working with cover elements. The design of interaction between objects from different covers is typically performed by substitutions of client environment parameters into templates supplied by the server cover (so called, statically coherent covers).

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 design of a coherent object space inside the cover, as well as design of other covers, whose objects could interact with this cover and its elements.

At run time the covers represent an “environments”, in which objects exist and communicate with each other. The cover controls the life cycle of objects, that is creation and deletion of objects, which are the features of “object factory”. Also, it organizes the communication with the environment; i.e. objects from this cover as well as elements of other covers. The cover controls all aspects of all objects contained in. For example, it provides the discipline of access to the objects. To get an access to the cover’s context one needs to make a respective request to the cover. It is important to notice, that covers act transparently for client object, as well as for server one. So, both client and server objects should be placed in some covers. All object communication problems are solved by means of conformation of appropriate covers.

Each cover contains so called communication kernel. It serves to support objects communication inside the cover and with the objects from other covers. Communication kernel is a part of cover, but it is not necessary for the kernel to be an object itself. The kernel provides a basic mechanism of object interaction. It determines mutual location of objects, which take part in the interaction (local or remote). And according to the objects location one or another communication scheme is used. This technique allows object communication to be more efficient in the case both client and server objects are located in one and the same address space. In this way, the cover kernel provides location transparency of the objects. Objects communicate with the kernel by means of a special mediator, called Interface Object (IO). Due to the mechanism of late binding, IO can be used in the transparent manner.

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 joined in a federation in order to provide common name space for search [ANSA93].

The cover contains some service objects, which support application objects functionality. It is important to note, that applications do not call these objects explicitly. Such additional cover’s objects form the set of services (e.g. concurrency service, statistic service, persistency service, etc.). In this way, any object supported by the cover turns out to be included in some context. The context consists of different services. The content of the context, as well as the relations between application objects and context elements can be changed by the cover at run time.

It is obvious, that the assumption of orthogonality of services helps to simplify both the development of the services at the design stage and schemes of their usage at run time. The assumption of orthogonality allows to work with a service or with a subset of services separately from the others at both stages. Also, the inclusion of new services into the cover (upgrading) at the design stage can be performed smoothly.

The following example demonstrates some simple covers and their usage. In the context of the cover approach, we assume that technique of late binding is used. It can be performed by means the following operations (for notation we use CORBA IDL language):

```
A get_objectA_ptr (in string object_name);
void release_objectA_ptr (in A obj);
```

Execution of these operations is controlled by the cover, in which the client is located. Operation *get_objectA_ptr* returns the pointer to the server object of type A, identified by the string parameter object_name within the cover A_cover. This is the only way the client can get reference to a server object (of type A). Besides, the client will have to release this reference after the use by means of *release_objectA_ptr* operation.

Such technique gives the cover the opportunity to substitute object reference. So, the operation *get_objectA_ptr* return reference to a mediator (IO), rather than reference to the server object itself.

The simplest example of A_cover is an object whose methods implement basic functions of Naming Service. Also, it provides the following two methods (we assume that the cover works with the objects of type A only):

```
status get_object (in string name, out A result);
status delete_object (in A arg);
```

To arrange objects of type A the cover can use an array M of pairs “object name - object reference”. The method *get_object* returns the object reference, which is found by name in the array M. If the name is not found, new target object is created by the cover and reference to it is returned. In the last case, a new record corresponding to the new object is added to array M. The method *delete_object* deletes object, pointed out as a parameter, as well as appropriate array element.

So, at the beginning, the client cover method *get_objctA_ptr* invokes the method *get_object* of the cover A_cover. Then, an IO corresponding to the returned object reference is created. Finally, the client gets the reference to this IO, rather than reference to the target object of type A.

3. Cover implementation by means of Interface Object technique

The described cover model must fulfill the following requirements:

- system services of the cover must be orthogonal;
- all data, the cover and it's elements, must be strictly typed;
- the cover mechanism must be added in the transparent way for both client and server objects;
- the cover must provide generic templates described the work with it's elements;
- the cover must give opportunity to dynamically customize the schemes of working with it's elements.

To implement the cover model the extended metaobject control technique is proposed to use. The metaobject control technique is based on the following principles. The extension of application semantics with additional semantics and the interaction of objects from different covers are performed by means of special invocation execution, rather than due to changes in client or server objects. Additional method calls are added to the client call of a server object (called *target object*) transparently for the user. Cover's objects which implement these additional methods are not visible for the user as well. These objects, called metaobjects, form the metaobject context (*metacontext*) of the application. Also, groups of semantically coherent metaobjects are called *metaservices* (e.g. persistency, concurrency, access authorization services).

We will say that by means of metacontrol technique application object is *bound* to the metaobject environment (*metaenvironment*) of the cover which controls this object. Using covers to extend application semantics is called *metaextension*. The process itself of the extended execution of application call to target object will be called metaobject control (*metacontrol*).

The cover approach also takes into account that a program system may change after one has started using it. The described technique supports the evolution of the system, providing that adding new metaservices to the cover's metacontext (*metaupgrading*) was as smooth as possible. Addition of a new orthogonal metaservice does not require to update already existing metaservices.

Also, metacontext itself and schemes of working with it may change at run time in both client and server covers. But these changes do not affect the visible interface of server object.

It is proposed to implement the metaobject control in a unique, highly independent of the controlled applications way. At the same time, objects forming a metacontext can be a subject of metacontrol themselves; i.e. the system conforms to the principle of reflection. That is, the metacontext may contain not only objects specially designed for this purpose, but also objects designed earlier which, like application objects, know nothing about metacontrol. The schemes of working with such metaobjects can be fixed by means of technique of metaextension as well.

However, the inclusion of objects into the cover rises the well known problems concerning efficiency of implementation. The point is that traditional method of binding object to metaenvironment leads to appearance of nested layers of interpretation, which is the main source of inefficiency in such reflective systems [MMAY95].

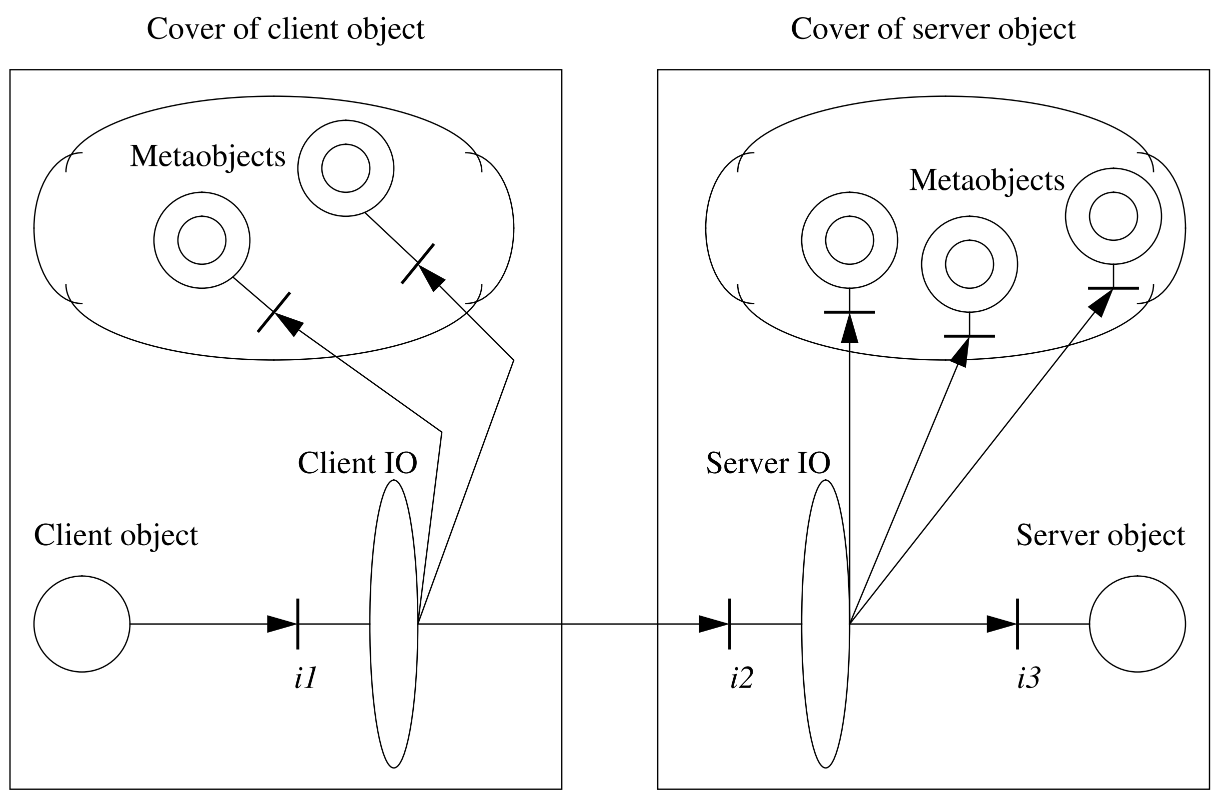

Figure 1. Metacontrol organization scheme.

In the context of the cover approach we offer the method of optimization of reflection, based on a redesign of the mechanism of binding object to metaenvironment of the cover. 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. It is IO, who binds object to metacontext and organizes additional metaobject calls. IO can be included into either the client cover (client IO), or server cover (server IO), or both.

Figure 1 illustrates the scheme of objects interaction by means of client and server IO. The real interface the client uses is interface of client IO (*i1*). Client IO calls methods of interface of server IO (*i2*). Finally, server IO actually invokes methods of target object interface (*i3*). The agreement on designations is the following. Application objects are defined by circles, IO by ellipses, metaobject by double circles.

It is obvious, that IO interface can be different from interface of target object. And it does not break the concept of client and server object transparent communication. Sometimes the difference between IO and target object interfaces can be necessary. The example is shown in the Appendix B.

In general, there are four variants of IO usage:

- client and server objects communicate without any mediators (metacontrol is not performed);
- only client IO is used; binding to metaenvironment of client cover is done;
- only server IO is used; binding to metaenvironment of server cover is done;
- both client and server IO are used; the metacontrol is worked out by metaobjects of both client ans server objects.

In case of remote invocation, since the *marshalling* of parameters according to some protocol is necessary, client and server IOs organize communication via the cover kernels. Each cover kernel performs all needed data transformation and makes actual data transmission to another cover kernel. At the same time, the basic mechanism of data transmission provided by the kernel can be extended by means of IO technique and appropriate metaobjects (for more details see section 7 and Appendix A). Since interface object and corresponding application object are located in a single cover, the communication between them does not require any data transformation and does not depend on mutual location of application objects. So, we may say, that a pair of IOs used on both ends of remote interaction provides the location transparency of application objects. That is, client and server objects can know nothing about where the other party is located. Appropriate IOs choose needed mechanism of communication itself.

4. Means of specification of binding object to cover metaenvironment

Remind our agreement that interfaces of all objects (in particular, IO and metaobjects) are described by means of CORBA IDL language. We do not bear in mind any concrete programming language and use an IDL-like syntax agreements to describe metacontrol. As it was already said, we assume the programing language to be strictly typed and, in general, object-oriented.

Another specification language, TL language, serves to describe variants of metacontrol schemes for different client and server metaenvironment. Metacontrol descriptions in TL do not depend on concrete application and programming environment. For each object type (type means interface specification in IDL) several types of client and/or server IO can be described in TL. Each IO has some set of parameters, which can be initialized and changed at run time. In this way, IO can be dynamically customized to some metaenvironment. That is, by means of parameters substitution the cover can change IO's binding with metaobjects at run time. Also, statically defined metacontrol schemes can be chosen dynamically. In general, metacontrol scheme means the sequence of metaobject invocations.

So, TL language is intended to describe binding object to metaenvironment. TL compiler has as input TL description of such binding, as well as IDL specification of all object interfaces used. The compiler generates code in a required target language.

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

4.1. Metacontrol specification example

Remind, that in IDL each parameter declaration must have one of the three passing modes:

- **IN** - input parameter;
- **OUT** - output parameter;
- **INOUT** - in-output parameter;

Without loss of generality, all methods are considered to be procedures in this example.

The example is as follows. Two objects O1 and O2 exchange information represented by strings. Object O1 calls on object O2 which implements an interface T with the method:

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

Let us consider the situation of remote communication between objects O1 and O2. In this case, the subject of metacontrol is 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 before sending and decompress the data received from the net.

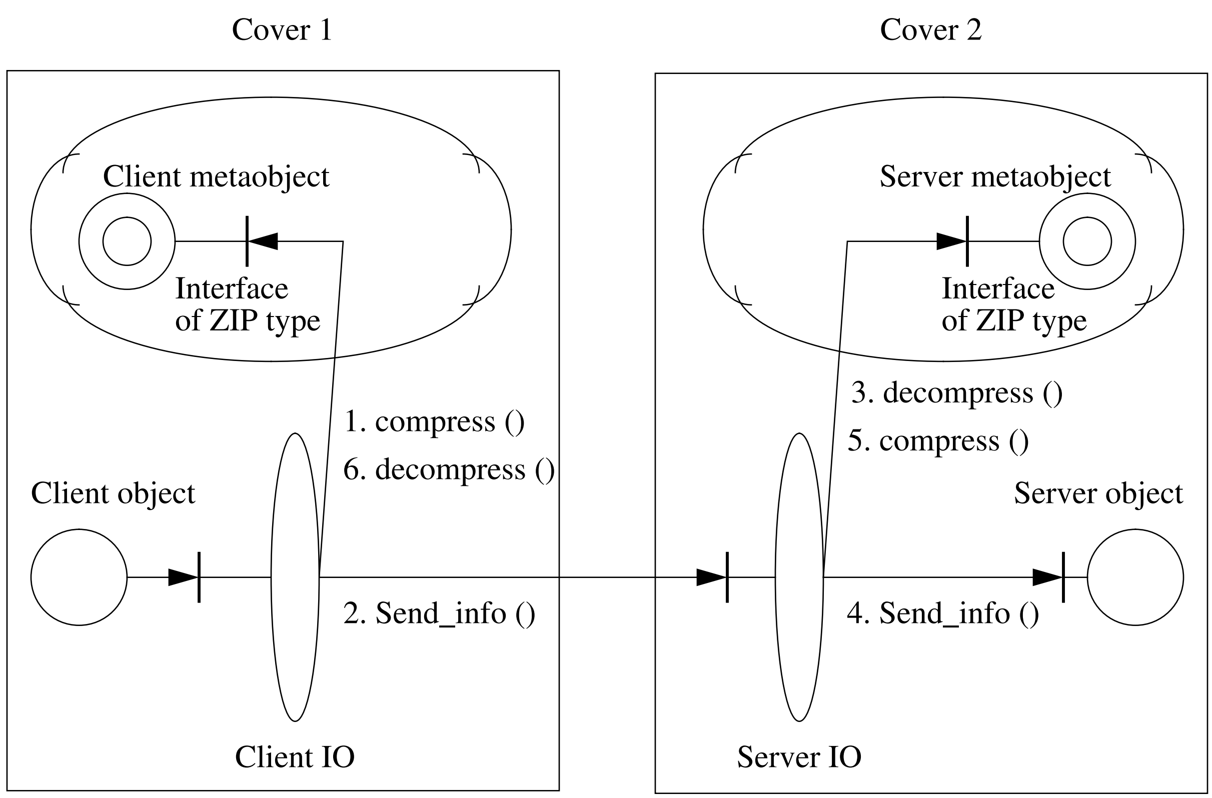

Figure 2. Metacontrol organization example.

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

```
compress (IN string info, OUT string compressed_info);
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 by means of TL language 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 a part of state of both client and server IO. The cover initializes it by a reference to a metaobject performing compression/decompression. For client IO, as well as for server one, implementation of method *exchange_info* contains two invocations on metaobjects and an invocation on target object itself. Fig. 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 server IO.

4.2. IO operation execution scheme

The example described above demonstrates the usage of only one variant of metacontrol, which performs the data compression/decompression in the transparent manner. But execution of these additional methods implies a consumption of computation resources. Therefore, it makes sense to provide another variant of metacontrol, when data are transmitted without changing. These two alternatives of metacontrol schemes allow client and server to consistently choose the variant of conforming metacontrol at run time. That is, the current metacontext can be dynamically changed.

The alternative scheme of metacontrol described above consists of a single invocation of method *send_info*. For the client IO:

```
{
    // server IO's method call:
    send_info (compressed_arg, compressed_res);
};
```

and for server IO:

```
{
    // target object's method call:
    send_info (arg_info , res_info);
};
```

The above metacontrol switching mechanism is implemented by means of the cluster technique (described in detail in section 4.7). In this way, the user needs to point out all possible variants of metacontrol which can be switched at run time, before compilation. 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. The cover makes decision on the choice of needed metacontrol scheme at run time. And exactly this scheme is executed during invocation of appropriate IO method. It is important to note, that one of statically specified metacontrol scheme can be replaced by another one at run time by client and server covers consistently.

4.3. Basic means for metacontrol description

The example described in section 4.1 illustrates that one needs to use temporary variables in metacontrol specification in order to store some temporary data between methods invocations on metaobjects. It implies the question about scopes of such variables. Method calls using one and the same variables are picked out in blocks that encapsulate such variables.

Thus, one should declare any metacontrol alternative (i.e. coherent statically defined sequence of object invocations) by means of block structures of proposed TL language. Blocks have the following properties:

1. Blocks can be nested.

2. Block may contain local variables declaration section at the very beginning.

3. Standard scope rules are used for variables encapsulated in blocks.

4. Some blocks may have labels in order to allow transition between them.

Block declaration looks as follows:

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

where every invocation consists of reference to object to be invoked, method name, and list of parameters used in invocation: object.method(par_1, ..., par_n).

Object reference can be omitted in description of IO's target object invocation only: target_method(par_1, ..., par_n).

Invocations and nested blocks in TL can be declared in any order. Nested blocks have the same structure as the outer ones. That is, local variables can be declared in nested blocks, besides, variables of all outer blocks are accessible according to standard scope rules.

4.4. Metaobject environment specification for applications

Object references contained in IO must be declared as a part of IO state. It should be done before one use them for metaobject method calls. In the example described in section 4.1 the following declaration are used:

```
OBJECT Zip_metaobject : ZIP;
```

It introduces IO data member (or part of IO state) - object reference *Zip_metaobject* of ZIP type. Besides, attribute (in the sense of IDL language) corresponded to data member appears in the IO interface declaration. The attribute is accessible from cover and has the same name and type as data member from above declaration. The attribute is dedicated to data member access, however client cannot use it. This attribute is accessible only for cover of the IO, more exactly for some cover's metaservice which manages conformation of IO and its metaenvironment and possibly metaenvironment of target object. That is, IO has two different interfaces: one visible for clients and another one that is not visible and is not accessible by clients. It is called *metainterface* and is dedicated to IO state manipulation. Metainterface is known to cover and is used for initialization during IO creation and for later changes of IO state. Considered declarations having OBJECT keyword together with declarations of cluster sets form IO metainterface and are discussed later in section 4.7.

Figure 3 shows both IO interfaces. Dotted lines designate manageability (the possibility of modification), and solid lines designate only use of element pointed by arrow.

At run time metacontrol is carried out by means of sequence of method invocations on some metaobjects. Object references to metaobjects are formed in discussed way. Cover containing IO can dynamicaly switch one metacontrol scheme to another by changing appropriate object reference (by dynamic altering of IO data member value).

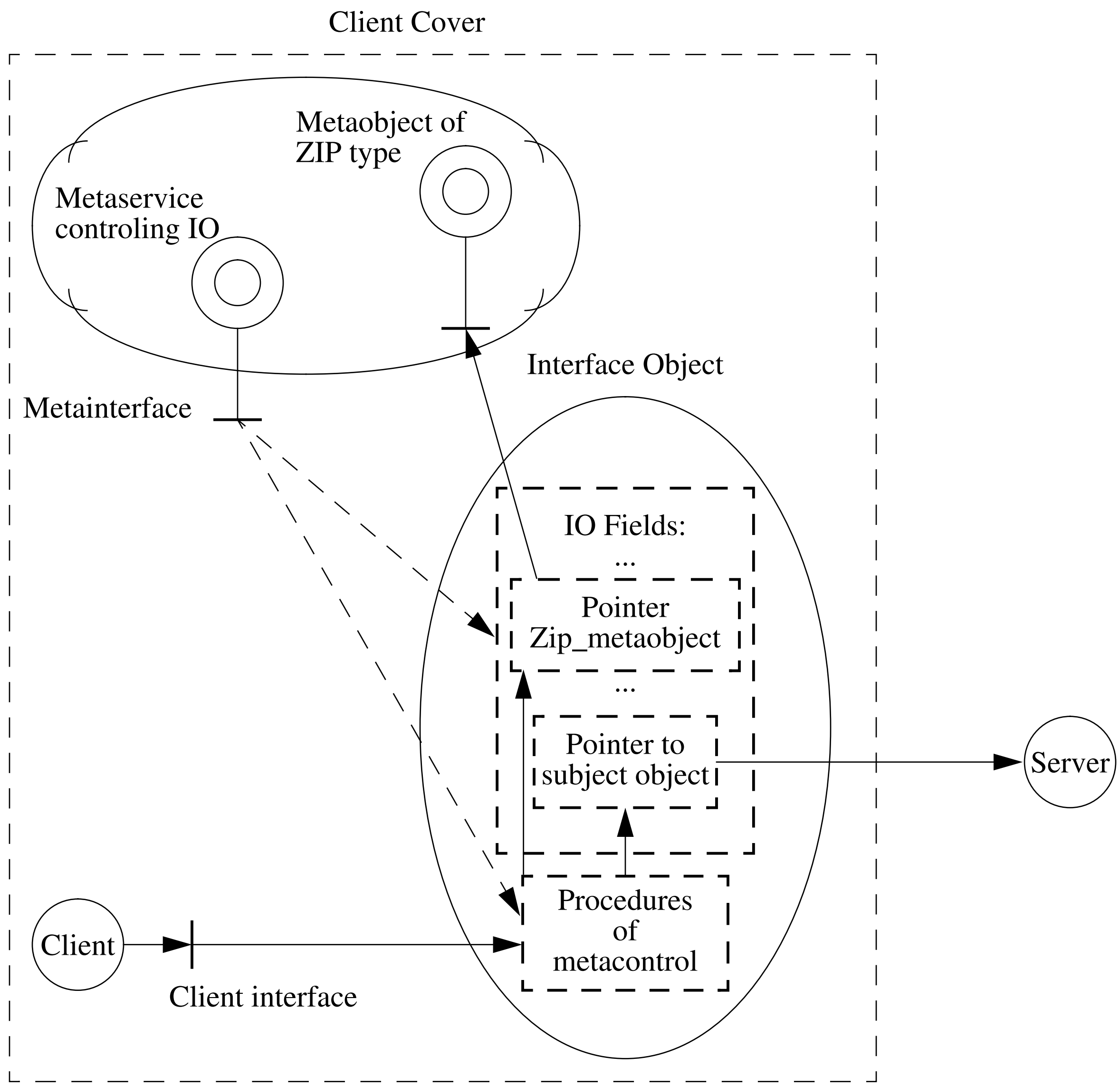

Figure 3. Scheme of work with interface object.

We can say, that collection of such declarations of object references used as IO parameters reflects some metaenvironment in which the IO could be potentially put by cover at run time. That is all metaobject's types that metacontrol of IO can use must be defined in these declarations.

4.5. Templates libraries

Metacontrol programming dealing with the same types of objects often leads to practically identical sequences of declarations. 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 objects (services). Libraries may correspond to services themselves and to services in the context of a certain type of a cover. In the last case cover refines service semantics use and affects possible use schemes of the service. Set of libraries corresponding to various cover services can be treated as an informational element of cover at design stage as it was said in section 2.

Semantics of a cover's service determines typical sequences of service method invocations. It is a basic idea for templates library construction. For example, access controlling service (concurrency service) having two operations (lock that acquires object access and unlock that releases object) can be used with the only sequence of service calls:

```
lock ( rw_flag ) ;
Work with managed object;
unlock ;
```

where rw_flag parameter sets needed lock mode - write (rw_flag = WRITE) or read (rw_flag = READ) one. Above sequence of calls can be declared as a template as follows:

```
LockPattern ( rw_flag ) < concurrency_objref > { ControlledOperations } IS {
        A {
            concurrency_objref . lock ( rw_flag );
            DO ControlledOperations WITH EXCEPTION HANDLER B;
        }
        B {
            concurrency_objref . unlock ;
        }
}
```

Identifiers used in the template have the following meanings:

LockPattern - template name;

rw_flag - template parameter, the name of lock operation parameter;

concurrency_objref - template parameter, the name of access control service;

ControlledOperations - template parameter, controlled operations;

A, B - block's labels;

WITH EXCEPTION HANDLER - keywords, they means the control is trasferred to a given block if exception is raised during controlled operations execution (here block B is used as an exception handler).

Thus, we described typical concurrency metaservice object that is to be binded with some target object. Obviously, specific binding description is a refinement of above template. Refinement consists of specific method parameters, object references, and sequences of calls (the place for which is reserved by parameters similar to ControlledOperations one) substitution. The last parameters kind allows, for instance, one to use template for several methods of target object having identical metacontrol. It is necessary in such case to pass target method call specification as value of ControlledOperations parameter in metacontrol definiton for every such method.

Let us consider example illustrating concurrency service template use. We assume there is an IO with the appropriate attribute declaration (see section 4.4):

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

Then method *some_operation (in SomeType arg)* that always modifies object's state can be metacontrolled in IO with the help of concurrency service template as follows:

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

It should be noted that keyword OPERATION used in above specification means target object's method invocation (in our case it is *some_operation* method) with the given IO parameters (in this example it is *arg* parameter). This keyword allows one to define identical metacontrol for group of methods in one specification:

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

4.6. Using templates compositions

At design stage of metacontrol to work with services of another covers programmer should be able to get generic templates from that cover and use them. It allows source code to be reused and programmer to know nothing about object call details of used services. Besides, presence of templates libraries gives possibility to one to orthogonalize (to make independent) various services in metacontrol descriptions. Programmer can abstract from details of method invocation in a given service and think in terms of refinements of templates compositions. Every template is interpreted now as a large-scale semantic unit of cover's service.

Let us consider now how one can bind two independent services to IO. For this purpose we continue our example in which *some_operation* method is wrapped with synchronization locks. We assume that IO has also another attribute referencing to statistics service of type Statistic:

```
OBJECT stat_obj : Statistic;
```

This new service has *begin_time* and *finish_time* methods dedicated to store start and finish time of some activity. Obviously, it is worth to put service template in the templates library:

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

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

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

So, *some_operation* invocation can be wrapped with concurrency control and statistics irrespective of details of implementations for both templates. In this order nesting provides lock-unlock service not only for controlled operation, but also for statistics service calls. That is a pair of objects - controlled server object and statistics service are protected. If it is desirable to protect only server object call nesting must be altered:

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

4.7. Using clusters

As it was noted above (see section 4.2), each IO operation might have several variants (schemes) of metacontrol defined statically, i.e. at the design stage. Obviously, IO should have mechanism allowing dynamic (at run time) switch of variants. One of possible solutions is to add some specific data member (or attribute in the sense of IDL) of enumeration type to IO. Changing value of the attribute one might change metacontrol scheme. What's more, attribute value change can be considered as an entire IO state change, but not only as a change of execution scheme of one particular operation.

Attributes permit execution scheme change for several operations and even for all IO operations at once. From this point of view we can say that a set of clusters for IO is provided. Each of clusters affects metacontrol scheme for one or more operations and has its own identifier. Set of cluster's identifiers makes values of enumeration type for correspondent IO attribute.

Let us continue statistics service example from section 4.6 in order to illustrate work with a cluster set. Besides of described in StatisticPattern template metacontrol scheme, we want to provide controlled operation execution without statistics collection. For this purpose 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, as well as StatisticPattern template used in them:

```
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;
        }
}
```

Keyword DEFAULT used in clusters set declaration points out that MakeStatistic variant is a default one. That is, after IO creation initial value of StatisticClusterSet attribute is MakeStatistic. Therefore, statistics service is used as it is indicated by first template until cluster is changed, i.e. until attribute value is changed.

Now, when one need to use this library's declarations in metacontrol definitions he/she should refer to it explicitly:

```
USE TEMPLATES LIBRARY StatisticLib;
```

Such library referencing leads not only to possibility of its templates use, but to automatic addition of StatisticClusterSet attribute to IO.

If declaration of IO attribute (object reference stat_obj of Statistic type) is similar to one from previous example (see section 4.6) metacontrol description for *some_operation* method looks like follows:

```
some_operation (arg) USE {
    StatisticCasePattern < stat_obj > { OPERATION }
}
```

Using StatisticClusterSet attribute, cover of IO may change current metacontrol scheme for a given operation at any time.

Note, that clusters allow one to use not only various metacontrol for the same methods. Also, it helps to make different groups of target object's methods be accessible to client at different time. Example illustrating use of clusters in this sense is considered in details in Appendix B. In this example access management for object encapsulating a file is described. In this case client IO contains three clusters. They are active when 1) file is not opened (only method for file opening is accessible); 2) file is opened for reading (reading, seeking, and closing methods are accessible); 3) file is opened for reading and writing (client can invoke reading, writing, seeking, and closing methods).

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 varied independently of each other. These attributes usage makes possible to manage orthogonal metaservices independently.

For example, we can bind concurrent access control metaservice (Concurrency service mentioned in section 4.5) to IO which has already been worked with statistics service. It leads to inclusion of ConcurClusterSet set of clusters and ConcurCasePattern template into service's template library besides of LockPattern template described in section 4.5. These templates define two alternatives for service use (similar to statistics metaservice) which define whether metaservice used (MultiAccess cluster) or not (NoConcurrency cluster):

```
TEMPLATES LIBRARY ConcurrencyLib {
    ... // description of LockPattern

    CLUSTER SET ConcurClusterSet {
        DEFAULT MultiAccess, NoConcurrency };
```

```
        ConcurCasePattern ( rw_flag ) <service_ref> { ControlledOperations } IS {
            MultiAccess :
                LockPattern ( rw_flag ) < service_ref> { ControlledOperations };
            NoConcurrency : DO ControlledOperations;
        }
}
```

Metacontrol description using both these metaservices should reference both appropriate libraries:

```
USE TEMPLATES LIBRARY StatisticLib;
USE TEMPLATES LIBRARY ConcurrencyLib;
```

Also, two IO data members are declared:

```
OBJECT concur_service : Concurrency;
OBJECT stat_obj : Statistic;
```

some_operation method's metacontrol using both metaservices might be defined as follows:

```
some_operation (arg) USE {
        StatisticCasePattern < stat_obj > {
            ConcurCasePattern (WRITE) < ñoncur_service > { OPERATION }
}
```

As one can see, this description of metaservices composition is similar to one used appropriate templates without clasters (see section 4.6). However, two independent IO attributes ConcurClusterSet and StatisticClusterSet are declared here. Every attribute can be managered by appropriate metaservice knowing nothing about other attributes and corresponded metaservice.

4.8. IO sequential refinement technique

Let us pay attention to sequence of metacontrol definitions described in the previous section. As one can see, at the begining statistics service was separately described and then Concurrency service was added. So, to add possibility of work with new service the description of metacontrol specification must be rewritten. However such step-wise refinement seems very natural during real metacontrol development: IO specification is the result of adding work with metaservices in sequence, rather than is fixed in one step. 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 refinment has its own identifier (name) and is a generic description for some part of metacontrol. So, description of the work with statistics service (example from section 4.7) can be considered as its 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 spicification 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 futher refinements. Next refinement uses descriptions from first one (it is referred by keyword REFINES) and adds work with metaservice Concurrency through parameter substitution:

```
Interf {
        // Second refinement of IO
        USE TEMPLATES LIBRARY ConcurrencyLib;
        OBJECT ñoncur_service : Concurrency;

        SERVER SecondVersion
            REFINES FirstVersion < ConcurCasePattern (WRITE) <ñoncur_service> >;
}
```

Obviously, such separated IO definition is not only convenient for IO development representation, it also simplifies possible redesign of metacontrol, because usage of different metaservices are localized in different refinements.

Besides, IO refinement process makes possible to separate development of client IO at the design stage in server and client covers, respectively, in quite natural way. Let us continue example from section 4.1 to illustrate this statement. The example concerns of compression/decompression which can be added to interaction between two remote objects.

Interface T in TL from the example contains metacontrol for client and server IOs. Remember that similar metacontrol descriptions are contained in server design stage cover (it contains declarations for server object of type T). So, only metacontrol from server IO can be fully defined in this way. As for client IO server cover's defined metacontrol is insufficient to it in the most cases. Server cover defines only those metaservices in client IO that are necessary for conforming interaction with server IO (in our case it is a metaobject for data compression/decompression on the client side). In general, client IO metacontrol refinement takes place in client design stage cover. Client cover can both refine metaobjects use and add new orthogonal metaservices used in this client cover. Such refinement and addition in client cover might be done step by step.

First refinement of client IO is described on the server side and called ServerVersion.

```
T {
    // First (server) version of metacontrol description for client IO
    CLIENT ServerVersion <compr_decompr_type, compress_op, decompress_op >
    IS {
        // IO's data member which type will be defined in client cover
        OBJECT metaobject : compr_decompr_type;

        exchange_info (arg_info, res_info) USE {
            variable compressed_arg : string;
            variable compressed_res : string;

            Zip_metaobject . compress_op (arg_info , compressed_arg);
            // server IO's method call:
            send_info (compressed_arg, compressed_res);
            Zip_metaobject . decompress_op (compressed_res, res_info);
        };
    };
};
```

This refinement contains three parameters with the following meanings:

- *compr_decompr_type* - metaobject compression/decompression type;
- *compress_op* - metaobject method for data compression;
- *decompress_op* - metaobject method for data decompression.

Client design stage cover sets specific metaobject type and its method names as first refinement of the generic IO description received from server cover:

```
T {
    // Second version of metacontrol description for client IO
    CLIENT FirstClientVersion
        REFINES ServerVersion <ZIP, compress, decompress >;
}
```

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

```
T {
    // Third version of metacontrol description for client IO
    CLIENT SecondClientVersion { some_statistic_pattern {} }
        REFINES FirstClientVersion
    IS {
        exchange_info (arg_info, res_info) USE {
            sîme_statistic_pattern {
                FirstClientVersion::exchange_info (arg_info, res_info) }
        };
    };
};
```

Third (second on the client side) refinement has parameter some_statistic_pattern that should be refined later with some template describing usage of statistics service. It is worth to note the metacontrol definition for *exchange_info* method in this example uses the same method declarations as in previous refinement (*FirstClientVersion::exchange_info*): statistics service template wraps existing metacontrol. In other words refinement technique allows one not only to substitute parameters in earlier defined refinements, but to extend metacontrol with new orthogonal metaservices too.

Last refinement has all parameters resolved. For this purpose object reference to metaobject of type Statistic is declared and StatisticPattern template is passed as parameter of previous refinement:

```
T {
    // announcement of statistics server’s library (from client cover) use
    USE TEMPLATES LIBRARY StatisticLib;

    // IO’s data member that is reference to statistics metaservice
    OBJECT stat_obj : Statistic;

    // Final refinement of client IO description
    CLIENT
        REFINES SecondClientVersion { StatisticPattern <stat_obj> };
};
```

This refinement cannot be refined further since it is not named.

TL language compiler treats refinement as finished one if it does not have parameters and it is not refined with another refinement. Whole sequence of refinements including last one defines resulting IO contents.

5. The cover's implementation

We deal with covers at both the design stage and the run-time stage. Within the object model framework, at both stages the cover is an object implementing a certain external IDL interface. For design stage covers, the external interface is understood as a collection of methods intended for future interaction between covers and other objects contained within them. For run-time covers, the external interface consists of methods destined to coordinate the work of objects of different covers. Also, the run-time covers provide an internal interface which is available only to their own objects.

Each design stage cover is targeted to the development of the contents of all run-time covers of certain type, and of mechanisms of operation with internal objects of these covers, as well as with objects of other covers. One of the most important tasks being solved by these covers during the system design is to provide a set of various metacontrol schemes for possible objects of the respective run-time covers. Besides, it is necessary to fix a set of statically named objects, i.e. objects of known types which can be referred to by names during the design stage. Mechanisms for run-time creation of objects also need to be developed. Such mechanisms must be used further to create objects before operating on them.

The external interface of the design stage cover contains methods intended for:

- provision with information on the corresponding interface of run-time covers;
- description of typical mechanisms of interaction with objects of the corresponding run-time covers in the form of a collection of templates libraries (see section 4.5);
- description of possibilities for dynamic change of the metacontrol for objects in run-time covers (i.e. for a change of a cluster or of IO's data members - see section 4.7).

The external interface of a run-time cover provides a possibility to work from other covers with a certain collection of services of the cover. A part of these services may be intended to control the work with the cover's objects of given type. In particular, such services may control the life time of the objects (creation and deletion of objects). Another group of services may be needed to support the work of the cover's internal objects, as well as to provide information on cover's contents.

Perhaps, the most important of such services is the naming service. It is the only service which is considered to be standard and which must be supported by all kinds of covers. To standardize operating with this service, we fix a corresponding set of methods as the base cover's interface which all other run-time covers interfaces must inherit from. Below follows a description of this base interface in IDL:

```
module Covers {
        typedef string Name;
        typedef sequence<Name> CompoundName;
        struct Binding {
            Name binding_name;
            Object binded_obj;
        };

        typedef sequence<Binding> BindingList;

        interface SearchIterator {
            boolean next_one (out Binding b);
            boolean next_n (in unsigned long how_many, out BindingList bl);
            void destroy();
        };

        interface BaseCover {
            void register (in Name n, in Object obj);
            void unregister (in Name n);
            Object simple_resolve (in Name n);
            Object compound_resolve (in CompoundName n);
            void local_search (in unsigned long how_many,
                                in Name search_pattern, out SearchIterator bi);
            void full_search (in unsigned long how_many,
                                in Name search_pattern, out SearchIterator bi);
        };
};
```

The above interface assumes objects to be identified either by simple names (Name) comprising of a single string, or by compound names (CompoundName) defined as a sequence of strings. Simple names are used in the following methods:

- register - registration under the designated name, of either the cover's internal object, or an external object (including external cover);
- unregister - deregistration of previously registered object;
- simple_resolve - resolving the name within the given cover according to the cover's rules (using cover nesting and federations as described further in section 6).

The two following methods are intended to arrange the search of name by a pattern; to navigate over the resulting list of associations "object-name", a search iterator (SearchIterator) is produced:

- local_search - a search within the given cover only;

- full_search - a search over the total space of associations "object-name" which are available from the given cover (according to the same rules as with the simple_resolve method).

Besides, a *compound_resolve* method is provided for resolving a compound name according to the standard rules (each component but the last one names a cover the rest of the name is to be searched in).

Thus we supply a programmer with an IDL specification of the *BaseCover* interface, as well as its implementation in a language being in use. In addition to this base mechanism, some cover type specific methods can be added by the user. Also, the implementation of base methods can be rewritten. Say, the functionality of the *simple_resolve* method can be extended by the will of the user so that on taking a name argument from a specific set of reserved names it creates a certain kind of object instead of making a search. Hence, a new object creation facility can be added to a cover without even adding new methods to the cover's interface.

6. Naming facility extension

Though a search by compound names can be applied as well to covers other than that in which the search was initiated, however the user of this facility is required to know a precise way in the cover's naming graph. In order to utterly simplify the usage of naming facility, some additional means providing transparency across the set of covers is necessary.

A possible way of implementing such a transparency is using a *hierarchy* of covers; i.e. covers can be nested like blocks in programming languages do. Similarly, when a simple name can not be found in a cover the search continues through covers that contain this one. In other words, the same standard visibility rule is applied as that used for variables within nested blocks.

A more complicated, however much more flexible way of extending naming facility is the usage of *federations*. Federation is a group of covers which allow each other to access to their spaces of object-to-name bindings. A simplest form of federation is that in which all members have an unrestricted access to the space of all other members of the federation. In a more general case covers can restrict other members' access to their space as well as their access to spaces of other covers of the federation; thus, the total search space reduces so as to accelerate the search. To specify these restrictions one can use either export list which defines what in this cover is accessible from the other covers, or import list which define what and where is visible from this cover.

Recall that both naming extension mechanisms mentioned above are used in a full_search method of the cover's base interface. The search over federations or enclosing covers has a lower priority than the search in the cover it was initiated in. In other words, a name is resolved first in the cover's local naming service and only if the resolution fail it will be continued according to mechanisms specified above.

Thus, with the help of the extended naming facility the covers allow their objects to resolve simple names against a distributed naming space as if they are local. To achieve this, the cover only needs to be adjoined in a federation with all covers being potentially interesting for its objects.

Note, however, that in some programming systems the client is not always an object. In this case, to unify the cover technique, particularly to provide a uniform usage of namings one might need to extend an object model.

For example, in the programming system OS UNIX / C++, routines run within processes starting from the function *main*. The usage of other functions which are not methods of objects is also allowed in this programming environment as the legacy of C language. However, the need of using object references and name resolution may occur in this functions as well as in C++ object methods. For the proposed technique to be unified, let us consider UNIX processes as a sort of objects. These are pure client objects without any interface. Any run time activity takes place either in these objects or in other objects invoked via object reference. *All* such objects must be ascribed to some covers. Thus, in the extended object model, any usage of object reference occurs in the frame of an object within a cover.

When a programming environment provides a guaranteed context (i.e. object, cover) for an arbitrary client job, one can speak about a unified way of using cover's internal interface. The methods of the unified interface are called in the same way from different objects. The execution of a method depends solely on the context; i.e. it is defined by mechanisms of the current cover. Thus, say, method *Object_resolve(in Name n)* of the internal interface uses for resolution of a given simple name the mechanism of the cover containing the calling object.

A substantial feature of cover internal interface is that the methods can be implemented as functions which do not relate to a particular object (if it is allowed by the programming language). For example, the above method is implemented in C++ as a function with the following signature:

```
Object *resolve (const Name &n).
```

It is convenient to use such way of calling when defining metacontrol. For example:

```
obj_ptr = RESOLVE (name);
```

Such call does not require an object reference created earlier. Rather, it executes according to the algorithm provided by the cover containing the interface object at run time. In particular, this facility can help solving IO initialization problem. That is, one can use such calls to initialize IO state (see section 4.4) during its initialization. To do it, one only needs to fix in such calls just several simple names of metaobjects at the project stage. And at the run-time stage, each cover in which such an IO is created must resolve these names possibly making use of federation mechanism and cover hierarchy.

7. Conformation of covers

As was mentioned before, each run-time cover interface contains, in addition to the base interface, also other interfaces for various services contained. These services may be designed specifically for manipulating certain groups of objects, say, objects of specific type. These interfaces can be added by inheriting specifications in IDL. Note that service interfaces also may inherit each other.

Each element of the object space is contained in a cover, which provides, in addition to object creation and deletion, also a metaenvironment required for these objects, and customizes interface objects included in it during their creation and work. Methods of different services can be used to control from the outside the creation and the deletion of objects, as well as for various kind of control for metaenvironment inside the cover.

To enable the control of the object's work in the frame of a cover, a special interface object method *BaseCover get_cover()* is provided, which returns a reference to the cover containing the target object of the given IO. It is worth noting, that during the further work with this cover's object reference, one need not know the exact type of the cover, that is the whole interface. Rather, only the interface of the needed service must be known. It is provided by the mechanism of inheriting interfaces.

One of the most important application of the described technology for manipulating covers is conformation of metaenvironments of different objects, which enables their interaction. This may reduce to conformation of covers containing this objects, which is conducted with the help of the corresponding services of these covers.

Recall the necessity to conform the behavior of covers in the example in section 4.2. In this example two different schemes of metacontrol have been described at the project stage for both server and client objects: one involved compression/decompression of data, while other manipulated data without changing them. For the first scheme, the metacontrol technology presumes that the related cover supplies the IO with the metaobject capable of making the required transformation. Clearly, the client and server IO can understand each other only when the enclosing covers can come to an agreement about which kind of metacontrol to use. That is, these covers should contain a service for conformation of metacontrol for the considered kind of objects. The conformation of metacontrol can take place not only at the appearance of a new client, in which case the client cover asks the server one for initialization of the metacontrol of the same type. A conformation mechanism can be provided, which can be used by both covers dynamically yet after the connection between the client and the server is established. The need of this type can appear in either cover, say, when the conditions of existence of IO within the cover change. As an evident example of a situation when the dynamic conformation is needed consider the migration (translocation) of either of the two objects.

A more complicated example of cover conformation is described in Appendix A. The example is about the usage of covers for adding new, more efficient mechanisms to arrange the interaction between objects in the frame of CORBA technology (see also section 8 below). In particular, we describe a method of communication between objects from different processes of a single workstation (OS UNIX is considered) over the shared memory with the help of the client and server metaobjects. Surely, for the normal joint work of the client and server IO, a coherent customizing of the IO metacontrol provided by special services of the enclosing covers is needed here as well.

8. Conformation with CORBA technology.

Below we discuss how the above mechanism of working with covers can be arranged and how to use the metacontrol relying on CORBA - the widely-known architecture for communicating objects in distributed systems. Note that the interface specification language IDL we are using now was borrowed from CORBA. In the modern CORBA standard are defined both IDL itself and its mapping onto programming languages C and C++. Hence, the application of cover mechanisms in the frame of CORBA implies, besides the usage of IDL itself, also much of the features of interfaces defined with the help of TL language. In particular, the mapping fixes the properties of object references with respect to the programming language needed, including such features as:

- inheritance;
- memory management, in particular wrt passing parameters to methods;
- using mediator objects as A_var representing an object reference of interface type A.

Clearly, even just these objectives have a significant impact on the implementation of interface objects whose great deal of functionality consists of manipulating object references to various metaobjects.

In addition, the CORBA architecture presumes the usage of mediator objects as object references. On the client side this mediator is called stub, on the server side it is called skeleton. The only purpose of these mediators is to provide the location transparency. That is, depending on the location of the client and the object requested, these mediators provide selection of the necessary method call managing mechanisms defined in ORB (Object Request Broker). As one can easily see, within the CORBA-related metacontrol the cover's communicational kernels (see section 2) can be based on such brokers.

Note that the CORBA mechanism for managing object references presupposes using a kind of late binding of objects. Hence CORBA applications intending to make use of metacontrol should arrange interconnection always via legal CORBA object references (that is one should not strive to improve efficiency by calling local object methods directly via well-known pointer). To say more, as ORB and produced by the IDL compiler mediators (stubs, skeletons) are not proper parts of the application (they are in fact parts of system environment), one can arrange metacontrol without updating application, by expanding the system environment, even if the application was not intended for the use of metacontrol. What one only needs is that various application objects communicate with each other and with the other application' objects by means of legal object references. As in this example with CORBA, one can see that in general case, applications designed for a certain system environment can be wrapped transparently (from the applications' point of view) by means of cover technique, provided that the system environment ensures the late binding due to the usage of mediators supplied by the environment.

As for CORBA itself, the question remains, how does IO which is also a mediator between client and server, relate with stub and skeleton.

For the beginning, consider 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 pass via the skeleton as a mediator. As one can easily see, both the server IO and the skeleton have very close generation schemes and occupy the same place in operating with target object. Hence, on the server side, these two kinds of mediators can by unified in the sense that the functionality if IO can be added to the skeleton.

As for the client side, the relation between stub and the client IO is somewhat more difficult to define. As a matter of fact, the 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 from other objects. Furthermore, the client often does not use stub as an object (that is, client doesn't call methods of stub). Instead, it is used as an object reference being passed to another object. In this case there is no need in using client IO at all.

Thus, in an attempt to unify stub and IO on the client side in a single mediator as we did on the server side, we would encounter the following problems:

- losing the ability to control the life time of the IO;
- having no ability to select needed IO for a given client;

- having too complicated stub when it is not used for method invocations and thus all IO specific methods and data are pure overheads.

So, on the client side, the stub and the client IO can not be unified. As the need of using IO gets apparent only with the first invocation via stub, the functionality of stub should be extended in the following way: 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 to the particular client, optionally conforming it to the server cover (i.e the cover may have the conformation service for IO types). Surely, many different algorithms for associating IO with stubs can be provided within the cover. For example, the stubs of the same type can use the same instance of IO, or the scheme “1 stub - 1 client IO” can be used instead.

After the stub receives a reference to the associated client IO from the cover, all method calls including the first one are passes to the IO. Surely, if the cover knows that a given client working with a given server does not need using a metacontrol, then the IO reference will be nil and the stub will simply pass the call further to the server (where it can be yet caught by a server IO).

Note that the location transparency provided by stubs and skeletons in accordance with CORBA standard was considered by us as a base functionality of IO. Surely, when the CORBA mediators are unified with IOs, the latter can provide the transparency either using the stub-skeleton features or adding new ways to arrange data transfer due to the usage of the metacontrol technique. For example, this can be brought about by the coherent usage of metaobjects providing the usage of the shared memory as described in Appendix A.

Thus, the described technology for expanding functionalities of stubs and skeletons allows to arrange a symmetric management of client and server IOs by means of covers. The covers themselves manage the creation of both types of IOs and have an opportunity to select the necessary type (version) of both client and server IOs. The described approach expands the base mechanisms of object invocation provided by CORBA. On both client and server sides, they include, generally speaking, nothing but a support for the remote object invocation. The mechanism of covers and interface objects provides a possibility not only to modify the behavior of an application object on the server side, but also to (equally!) distribute the additional states and functionalities between the server and client parts. Thus there arises an opportunity to divide semantically the responsibility of client and server sides for different aspects of managing application, which simplifies the design of the application and makes it more efficient.

It should be noted, that the technique of covers is brought to the CORBA model not at the expense of changes in ORB implementation, but rather by means of conforming the compiler of IDL to the implementation language with the compiler of metaspecifications (see section 1).

With the implementation of the cover technique on the base of CORBA, a question also arises about the interrelation between the cover metaservices and the standard CORBA services [COSS96].

As it was noted earlier, the only metaservice we have fixed as a base one, i.e. which must be implemented in all covers, is the naming service. As is generally known, the CORBA standard defines the interface CosNaming::NamingContext of naming service objects and describes the main features of its semantics. In order to preserve the compatibility of the cover naming service with the CORBA one, the latter is inherited by the main cover interface BaseCover. Thus, the base cover interface gives an opportunity to take cover as a CORBA naming context. In addition, the cover supports also a different from CORBA naming mechanism suggested earlier.

As for the rest of the standard CORBA services, the metacontrol technology allows to use them along with any other services both as cover metaservices as well as components of user applications. We do not regulate anyway such a usage which remains completely to the judgement of the programmer who designs an object system in the frame of the cover model.

References

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

[Cahill96] Vinny Cahill. An Overview of the Tigger - Object-Support Operating System Framework. SOFSEM'96: Theory and Practice of Informatics, Lecture Notes in Computer Science, volume 1175, Springer-Verlag, Berlin/Heidelberg, November, 1996, p. 34-35. (ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg-102.ps.gz)

[Chi96] Shigeru Chiba. OpenC++ Programmer's Guide for Version 2. Technical Report SPL-96-024, Xerox PARC, 1996. (http://www-masuda.is.s.u-tokyo.ac.jp/openc++.html)

[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 Position Paper from Xerox Parc.

[Klein96] Jan Kleindienst, Frantisek Plasil, Petr Tuma. Lessons Learned from Implementing the CORBA Persistent Object Service. OOPSLA'96 conference proceedings, San Jose, California, October 6-10, 1996.

[KLMKM93] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The Need for Customizable Operating Systems. In Proceedings of the Fourth Workshop on Workstation Operating Systems, pages 165--169. IEEE Computer Society Technical Committee on Operating Systems and Applications

Environment, IEEE Compuer Society Press, October 1993.

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

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

[MJD96] Malenfant J., Jacques M., Demers F.-N. A Tutorial on Behavioral Reflection and its Implementation. Proceedings of the Reflection'96 Conference, San Francisco, USA, April 21-23,1996

[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. p. 300-315.

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

[SoSt95] Soley R., Stone C. Object Management Architecture Guide. Third edition. John Wiley and Sons, Inc. 1995.

[Str94] D.Stroustrup. The C++ Programming Language 2nd Edition. Addison-Wesley, 1995.

[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. Proceedings of the International Conference on Configurable Distributed Systems, IEEE, Annapolis, Maryland, May, 1996. (ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg-100.ps.gz)

Appendix A

This appendix describes in detail the covers conformation example which was mentioned above in section 7. Two covers ClCover of ClientCover type and SvCover of ServerCover type are considered (see Figure 4). These covers are designed to establish more efficient communication (than ORB's standard mechanism) with server objects of some ServObj type by means of some metaservices. All server objects of this type work in SvCover cover that can be referred thereby as a server cover. In its turn client is located in ClCover cover (client cover). If client and server objects are resided in different processes in the same computer, special metaobjects are generated by the covers. These metaobjects play the role of mediators organizing data (parameters and results) transmission by means of shared memory segment which is common for client and server processes. This way of data transmission is often more efficient than standard mechanism the ORB uses.

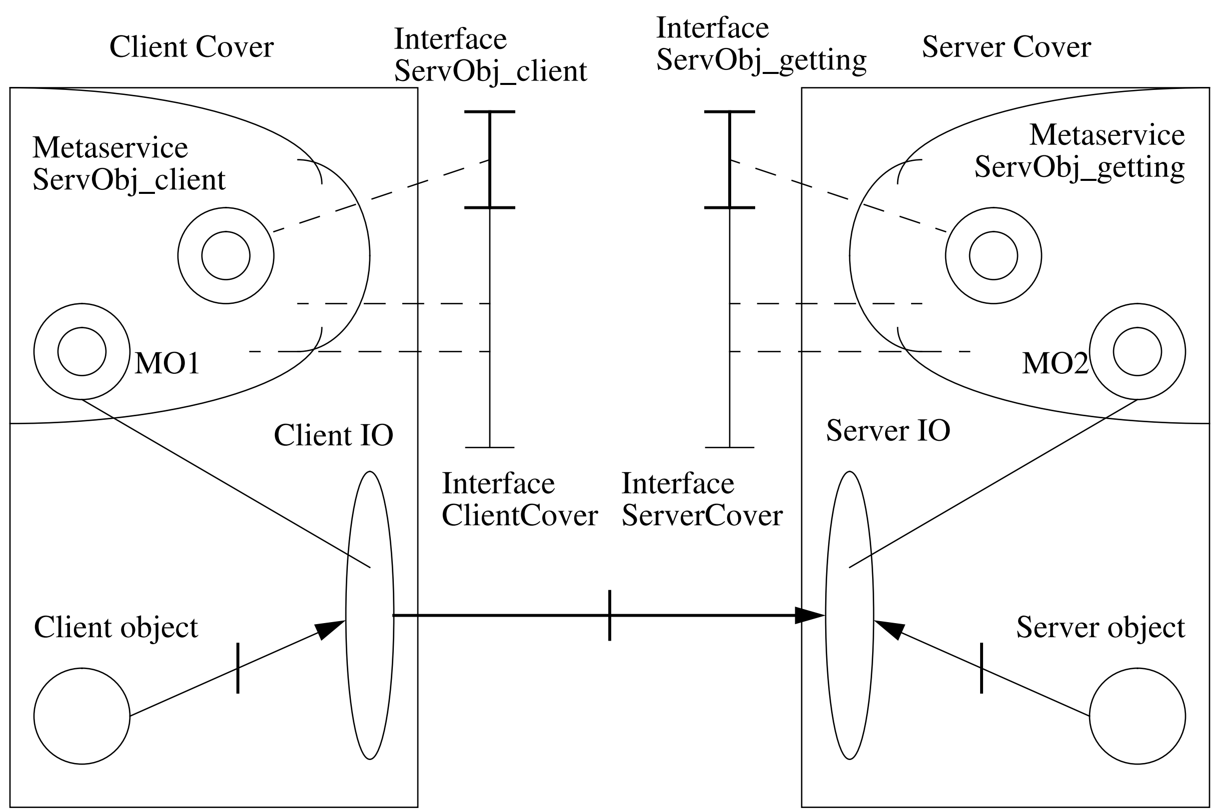

Figure 4. Covers conformation for organization of interconnection via shared memory.

The part of covers interfaces IDL specifications significant for this example is as follows:

```
// for server cover
interface ServObj_getting {
        ...
        ServObj use_optim_connection (in ServObj_IO client);
};
interface ServerCover : ..., ServObj_getting, ... {
        ...
};
// for client cover
interface ServObj_client {
        ...
        void use_shared_mem (in ServObj_IO client, in long sh_mem_segm);
};
interface ClientCover : ..., ServObj_client, ... {
        ...
};
```

So, the server cover includes ServObj_getting service that is responsible for creation of objects of ServObj type. One of the service's methods (*use_optim_connection*) allows the server cover to choose the most efficient way of communication with client object (that is, with appropriate IO) pointed as parameter of the method. In its turn, client cover has ServObj_client service which controls interface objects of ServObj_IO type. The method *use_shared_mem* of this service customizes the metacontext of IO pointed as the first parameter to use the shared memory segment (its identifier is provided by the second parameter) for communication with target object of this IO.

So we have the following algorithm of communication between client and server objects:

- client cover ClCover creates IO of ServObj_IO type;
- the object reference to server cover SvCover is searched out by means of naming service of client cover;
- IO being initialized calls *use_optim_connection* method of server cover; IO identifies itself in the parameter of this call;
- ServObj_getting service (which is used to perform the *use_optim_connection* method) of server cover creates both object of ServObj type and server IO to work with this object; further this service learns from object reference to client IO that client is resided in the same computer as server is resided; thereby server metaobject MO2 is created for organization of data transmission via shared memory;

- server cover creates a shared memory segment and passes identifier of this segment and reference to client IO into cover of this IO (the *get_cover* operation described above is used in server cover to get reference to client cover according to object reference to this IO), more precisely it calls *use_shared_mem* method of ServObj_client service from client cover;
- according to server cover's request client cover creates metaobject MO1 and customizes client IO to work via this metaobject;
- control flow is returned to server cover and than to client IO which gets reference to server IO created by server cover on previouse stage.

Thus during initialization client IO calls server cover and gets reference to server IO as a result of this call. Covers coherently customize environment of IO according to communication mechanizm chosen on the server side. Note, that IO does not take part in this conformation and client and server covers manage it themselvs.

Appendix B

This appendix demonstrates the example of internal interface use for organizing communication between client and server IOs (this means that interface of server IO is distinguished from interface of target object). Besides, this example shows the use of cluster mechanism (see 4.7.) for selection of some groups of interface methods that can be used by client at the certain time. The possibility to use any of this methods is determined by current state of client IO.

In this example we suppose that we have got some application dedicated to work with files. This application acts within the framework of some cover which could be realized (for example) as some directory of a file system. Here we are interested in application objects (contained by such a cover) which present separate files of such directory. The naming of all such objects is performed by means of base naming service of the cover (see section 5). By means of this metaservice client gets access to some needed file. In this example we do not consider other questions concerned such a directory - cover (creation, removing, logical binding of files etc.).

All objects - files implements the following IDL interface:

```
enum Mode {READ, READ_WRITE};
typedef sequence<octet> Data;
typedef long Status;

interface File {
        Status  open (in Mode use_mode);
        Status  close ();
        Status  read (in long byte_cnt, out Data result);
```

```
        Status  write (in Data info);
        Status  seek (in long byte_cnt);
};
```

Application objects implementing this interface *File* are dedicated to work with one client only which has (by means of one of these objects) a possibility to:

- *open* a session of file use; client is to choose the mode of work with this file: only data reading (READ mode) or writing and reading (READ_WRITE mode) (this mode can determine, for example, the way of data buffering during opened session);
- *close* a session;
- *read* needed amount of bytes from current position in file; when this operation is finished, current position in file is exactly the last position data was read from;
- *write* data to file (this operation can be performed without errors only if current session was opened in READ_WRITE mode) beginning from the current position in file; when this operation is finished, current position in file is exactly the last position data was written at;
- set the current position in file (with seek operation) according to offset relative to file beginning.

All methods mentioned in the example return exit status as a result (that is, value of Status type).

By means of cover approach we are going not only to control life time of File objects but to organize multiaccess for different clients (possibly remote) to such objects too. As it was pointed above, application is not dedicated to be used by some clients simultaneously. We can use interface objects to solve the issue.

We can use division of READ and READ_WRITE modes, already provided by the application. For clients work synchronization on server side we will use metaobjects with the following IDL interface:

```
interface Concurrency {
        Status  connect (in Mode use_mode);
        Status  disconnect ();
        Status  lock () ;
        Status  unlock ();
};
```

Objects implementing this Concurrency interface provide a possibility to work with target object either for set of clients simultaneously in READ mode or only for one client in READ_WRITE mode. To open session in needed mode *connect* method is used, *disconnect* method is used to finish previously opened session; *lock* and *unlock* methods are to be used for transit locking of the object only for a while of one method execution within the framework of the current session. We suppose that all calls from different clients are handled in different threads (i.e. in multithread mode), and so in the UNIX process such threads can be delayed if some client is already working in READ_WRITE mode with the same server object.

There is the following pattern for work organization with *lock* and *unlock* methods in templates library for Concurrency service (it is similar to example from 4.5.):

```
TEMPLATES LIBRARY ConcurrLib{
        LockPattern < concurrency_objref > { ControlledOperations } IS {
            A {
                concurrency_objref . lock;
                DO ControlledOperations WITH EXCEPTION HANDLER B;
            }
            B {
                concurrency_objref . unlock ;
            }
        }
}
```

But the problem is that original application objects assume work with a single client and such a server object contains current position in file for *this* client. It is obvious that in case of multiaccess this way of work is possible only in READ_WRITE mode (because in this mode one server can work only with one client at the same time). But when some clients simultaneously read data from the file, all these clients must work so as they had exclusive access to his file. To support such kind of abstraction (i.e. metacontrol transparency) is exactly one of the general tasks of cover approach. That is, by means of IO mechanism we should organize special state (position in the file) for every individual client. This state must be taken into account in any *read* operation.

So, to save current position in file we should have a global (accessible to all methods and persistent between methods calls) variable (let's name it *current_position*) of *long* type in client IO.

Client IO can be in one of three different states that determine its behavior and client's possibility to use its methods. The following three clusters correspond to these states:

- NOT_CONNECTED - this atate of IO means that appropriate client object is not in the session of connection with target object yet (session is not opened yet or is already closed);
- READING - in this state file is accessible for reading only;

- WRITING - in this state client works with object-File in exclusive mode and can modify it.

It is obvious that only part of target object methods is accessible in all these states:

NOT_CONNECTED - open;

READING - read, seek, close;

WRITING - read, write, seek, close;

As it was mentioned above (see 4.7.) attempt to call method not contained in current cluster cause exception raising and neither methods of metaobjects nor method of target object are called.

Of course in READING and WRITING clusters metacontrol of their common methods *read* and *seek* must be different. In READING cluster (unlike WRITING cluster) *current_position* variable must be used. In READING cluster *seek* method saves its parameter in this variable only and does not call target object. And *read* method execution in the same cluster performs (at first) setting of current position in file according to value of *current_position* variable and (second) data reading from this position. But it can not be implemented as two sequential calls of *seek* and *read* methods because between these two calls any other client can change the state of the target object according its own state. So these calls can be performed only as a single operation on the server side. Thus we need additional method of server IO dedicated to data reading in multiaccess mode. This method has all the same parameters as *read* method and one additional parameter for delivering of current position of concrete client. The signature of this method on IDL is:

```
Status   read_from_position (in long position, in long byte_cnt, out Data result);
```

Metacontrol for client and server IOs described above can be specified in TL language as follows:

```
File {
        SERVER IS {
            // specification of changes in server IO interface
            ADD METHOD Status read_from_position (
                            in long position, in long byte_cnt, out Data result);
            // specification of templates library use:
            USE TEMPLATES LIBRARY ConcurrLib;
            // global variable (=data member of IO) definition
            OBJECT concur_serv : Concurrency;

            // definition of metacontrol in methods of server IO
            open (use_mode) USE {
```

```
        concur_serv . connect (use_mode);
    };
    close () USE {
        concur_serv . disconnect ();
    };
    read (byte_cnt, result), write (info), seek (byte_cnt) USE {
        LockPattern <concur_serv> { OPERATION; }
    };
    read_from_position (position, byte_cnt, result) USE {
        LockPattern <concur_serv> {
            seek (position);
            RESULT = read (byte_cnt, result);
        };
    };
};

CLIENT IS {
    CLUSTER SET CurrentState {
        DEFAULT NOT_CONNECTED, READING, WRITING };

    // global variable definition
    variable current_position : long;

    // definition of metacontrol in methods of client IO
    open (use_mode) USE {
        NOT_CONNECTED :
                    OPERATION;
                    if (use_mode == ReadingMode) {
                            CurrentState = READING;
                    } else {
                            CurrentState = WRITING;
                    }
    };
    write (info) USE {
        WRITING : OPERATION;
    };
    close () USE {
        READING :
                    OPERATION;
                    CurrentState = NOT_CONNECTED;
        WRITING :
                    OPERATION;
                    CurrentState = NOT_CONNECTED;
    };
```

```
            read (byte_cnt, result) USE {
                READING : RESULT = read_from_position (
                                            current_position, byte_cnt, result);
                WRITING : OPERATION;
            };
            seek (byte_cnt) USE {
                READING : {
                    current_position = byte_cnt;
                    RESULT = 0; // means that method finished correctly
                };
                WRITING : OPERATION;
            };
        };
};
```

It should be noted that metaobject of Concurrency type used by some IO has access to methods of this IO's target object too. This metaobject must call *open* and *close* methods of target object (from metacontrol description follows that server IO never calls these two methods itself) because only metaobject knows when target object should be switched from one mode to another (such a mode switch takes a place if all reading sessions are closed and new changing (in READ_WRITE mode) session begins and vice versa).

Note also that in more complex case of multiaccess organization data needed for restoring the state of server to work with some client (in this example it is client's offset in the file) can have much more complex structure. In this case more complex data handling is needed too. And so it is more convenient to use separate metaobject instead of IO state for work with such data.

Besides, to organize multiaccess it can be not enough to use methods of target object for every separate client's state handling. For instance, if object-File do not have *seek* method for moving of current position in the file (i.e. this object can provide only sequential access to data in the file and work with strictly encapsulated own state containing current offset in the file) then we would directly change internal state of target object to set position in the file for the client. In this case we would use special metaobject working with target object's content similar to metaservice described in Appendix C which provide persistency metaservice.

Appendix C

In this appendix we describe the example of persistency [COSS96, Klein96] metaservice which shows special metacontrol variant when metaextension is impossible without source program texts modification.

In this example application is a set of objects implementing the following Store interface:

```
typedef long Status;
typedef long Label;
typedef sequence<octet> Info;

interface Store {
        Label save (in Info info_piece);
        Status read (in Label what, out Info result);
        Status remove (in Label what);
}
```

Any such object is dedicated to be used by one client as store of data of any size. For instance, such server object can be used for saving of data from other processes which have not ability to store large portions of information in own memory. We can consider process that controls getting of video information and working on device which produces such amount of data but has no own memory for data storing. Server object of Store interface can be locally used as mediator to work with dynamic memory too.

Store object methods are dedicated to *save* given data portion which is present as byte (octet) sequence. This method allocates a dynamic memory piece of needed size for given information. The identifier (label) of this memory is returned as a result of the method. The next (*read*) method is intended for reading of previously saved data. And the last method (*remove*) can be used to remove data when they are obsolete.

So such server object is a simple mediator to work with dynamic memory of the process for both local and remote access.

Metaextension in this example consists in organization of the work with Persistency metaservice. It is done with *minimal* (but *needed*) changes in source texts of Store_realization object that implements Store interface.

For Persistency service organization application objects' cover uses metaobjects implementing the following interface:

```
interface PM {
        typedef string PID;

        Status persist_save_anywhere (in Label elem_id, out PID pid_res);
        Status persist_save (in Label elem_id, in PID pid_arg);
        Status persist_restore (in PID pid, out Label elem_id);
        Status persist_delete (in PID pid);
}
```

Such metaobject allows to save in file (PID contains a name of the file) either entire state of the controlled (by this metaobject) *Store* object (elem_id = -1) or only one its element pointed by *elem_id* label. There are two methods with this functionality: *persist_save* method allows user to choose saving file (which is identified by *pid* parameter value) and *persist_save_anywhere* methods finds out saving place itself. The next method (*persist_restore*) restores object's state from the file (entire state or one element of the state). And *persist_delete* method removes the pointed saved data (file).

Note that this example shows possibility to organize persistency by means of metacontrol not only for entire objects state but for some part of this state too. Methods of object of Persistency type provide a possibility to save pointed portion of information and to restore this informatio separately in future. Of course the data label can be changed after restoring (new label is returned through OUT parameter of *persist_restore* method).

This example also illustrates very important capability of proposed metacontrol mechanism. It consist in possibility to change interface of target object visible to client (indeed, means client IO interface) as it was done for server IO. In this case for saving ability of transparent work of metacontrol (it means that user has possibility to work with client IO so if user has direct access to application object's interface) we should only add new methods to client IO interface with inheritance of all methods of application object's interface. It is obvious that new methods of client IO can be executed without any calls to target object, that is they can be consist in work with metacontext only.

So described metaextension of Store objects drives to extension of interface available for client. But client still can work with these objects without knowing anything about changes in interface.

Following code in C++ language illustrates some features of described in this example special kind of metacontrol. We suppose that objects implementing Store interface are instances of the following Store_realization class:

```
class Store_realization {
protected:
        Info *info_arr;
        long arr_length;
public:
        Label save (const Info &info_piece);
        Status read (Label what, Info *&result);
        Status remove (Label what);
};
```

So every object - instance of this class saves needed information in array of structures, each of which is used to point to one data portion saved in dynamic memory (value of Label type is ordinal number of element in this array). The data member *info_arr* points to such array. Long *arr_length* is its current length (this length can be dynamically changed according to some algorithm and dependent on current stored data pieces). All three methods of the class work with this array and do not have own state interesting for Persistency service. Certainly, state encapsulated in object (Store_realization class instance) is accessible from outside of the object neither for reading nor writing.

To work with objects of Store_realization type we work out the following class:

```
class StorePersistency {
protected:
        Store_realization *_target_obj;
public:
        // constructor
        StorePersistency (Store_realization *target) : _target_obj (target) {};

        Status persist_save_anywhere (Label elem_id, PM::PID &pid_res);
        Status persist_save (Label elem_id, const PM::PID pid_arg);
        Status persist_restore (const PM::PID pid, Label &elem_id);
        Status persist_delete (const PM::PID pid);
};
```

Object (instance) of this type has data member *_target_obj* which points to some object of Store_realization type. This pointer is set (via constructor's parameter) by cover during object creation. But accessible from outside Store_realization methods do not allow to work with state of instances of this class. To allow instances of StorePersistency class to access to internal state of Store_realization we should add to its class definition the next string (in C++):

```
friend class StorePersistency;
```

This declaration makes all data members of Store_realization objects accessible (for reading and writing) to all objects of StorePersistency type. This change is the only one needed to be done in source program of the application.

To finish the example we consider possible implementation of some StorePersistency class methods:

```
Status
StorePersistency::persist_save_anywhere (Label elem_id, PM::PID &pid_res) {
        // search where to save data
        ...
        pid_res = ...;
        return persist_save (elem_id, pid_res);
```

```
}
Status
StorePersistency::persist_save (Label elem_id, const PM::PID pid_arg) {
        FILE *file = fopen (pid_arg, “wb”);
        if (file == 0)
            return -1;
        if (elem_id < 0) (
            // saving of entire state of out object
            Info *cur_info = _target_obj-> info_arr;

            fprintf (file, “%d”, _target_obj-> arr_length);
            for (int i = 0; i < _target_obj-> arr_length; i++, cur_info++) {
                // writing of length and content of current element to file
                fprintf (file, “%d”, cur_info->length());
                fwrite (&((*cur_info)[0]), cur_info->length(), 1, file);
            }
        }
        else {
            fprintf (file, “%d”, -1);
            ...// saving of one pointed element
        }
        return 0;
}
Status
StorePersistency::persist_restore (const PM::PID pid, Label &elem_id) {
        elem_id = -1;
        FILE *file = fopen (pid_arg, “rb”);
        if (file == 0)
            return -1;

        long elem_cnt;
        fscanf (file, “%d”, &elem_cnt);
        if (elem_cnt < 0) {
            ...// one element restoring
}
else  {
            ...// entire array _target_obj-> info_arr restoring
        }
        return 0;
}
Status
StorePersistency::persist_delete (const PM::PID pid) {
        ...// removing of the file named pid
}
```