Object Oriented Interoperability

Dimitri Konstantas
University of Geneva[1]

Abstract. Object Oriented Interoperability is an extension and generalization of the Procedure Oriented Interoperability approaches taken in the past. It provides an interoperability support frame by considering the object as the basic interoperation unit. This way interoperation is based on higher level abstractions and it is independent of the specific interface through which a service is used. A prototype implementation demonstrates both the feasibility of the ideas and the related implementation issues.

1 Introduction

An important issue in today's large heterogeneous networks is the support for *interoperability*, that is the ability of two or more entities, such as programs, objects, applications or environments, to communicate and cooperate despite differences in the implementation language, the execution environment or the model abstractions. The motivation in the introduction of interoperation between entities is the mutual exchange of information and the use of resources available on other environments.

1.1 Procedure Oriented Interoperability

The problem of interface matching between offered and requested services has been identified by many researchers [3][9][14][15][16][17][20] as an essential factor for a high level interoperability in open systems. Nevertheless, most of the approaches taken in the past [9][14][16][20] are based on the Remote Procedure Call (RPC) paradigm and handle interoperability at the point of procedure call. We call this type of interoperability support approach *Procedure Oriented Interoperability (POI)*. In Procedure Oriented Interoperability support it is assumed that the functionality offered by the server's procedures matches exactly the functionality requested by the client. Thus the main focus of the interoperability support is the adaption of the actual parameters passed to the procedure call at the client side to the requested procedures at the server side. An example of this approach is the one taken in the *Polylith* system [16]. The basic assumption of the approach is that the interface requested by the client (at the point of the procedure call) and the interface offered by the server "fail to match exactly". That is the offered and requested parameters of the operation calls differ. A language called *NIMBLE* has been developed that allows programmers to declare how the actual parameters of a procedure call should be rearranged and transformed in order to match the formal parameters of the target procedure. The supported parameter transformations include coercion

1. *Author's address:* Centre Universitaire d'Informatique, University of Geneva, 24, rue Général-Dufour, CH-1211 Geneva 4, Switzerland.
E-mail: dimitri@cui.unige.ch. *Tel:* +41 (22) 705.76.47. *Fax:* +41 (22) 320.29.27.

of parameters, as for example five integers to an array of integers, parameter evaluation, as for example the transformation of the strings "male" and "female" to integer values, and parameter extensions, that is, providing default values for missing parameters. The types of the parameters that are handled are basic data types (integers, strings, booleans etc.) and their aggregates (arrays or structures of integers, characters etc.). The programmer specifies the mapping between the actual parameters at the client side and the formal parameters at the server side using NIMBLE and the system will then automatically generate code that handles the transformations at run time.

Whereas NIMBLE focuses in bridging the differences between the offered and requested service interfaces, the *Specification Level Interoperability (SLI)* support of the *Arcadia* project [20] focuses on the generation of interfaces in the local execution environment through which services in other execution environments can be accessed. The major advantage of SLI is that it defines type compatibility in terms of the properties (specification) of the objects and hides representation differences for both abstract and simple types. This way SLI will hide, for example, the fact that a stack is represented as a linked list or as an array, making its representation irrelevant to the interoperating programs sharing the stack. In SLI the specifications of the types that are shared between interoperating programs are expressed in the *Unifying Type Model (UTM)* notation. UTM is a unifying model in the sense "*that it is sufficient for describing those properties of an entity's type that are relevant from the perspective of any of the interoperating programs that share instances of that type*"[20]. SLI provides a set of language bindings and underlying implementations that relate the relevant parts of a type definition given in the language to a definition as given in the UTM. In SLI the implementer of a new service will need to specify in UTM the service interface and provide any needed new type definitions for the shared objects and language bindings that do not already exist. In doing so the user will be assisted by the *automated assistance tools* which allow him to browse through the existing UTM definitions, language bindings and underlying implementations. Once a UTM definition for a service has been defined the *automated generation tool* will produce the necessary interface in the implementation language selected plus any representation and code needed to affect the implementation of object instances. This way the *automated generation tool* will always produce the same interface specification from the same UTM input. However SLI can provide different bindings and implementations for the generated interface allowing a service to be obtained from different servers on different environments, provided that they all have the same UTM interface definition.

A similar approach to SLI has been taken in the *Common Object Request Broker Architecture* (CORBA) [14] of the Object Management Group (OMG). The Object Request Broker (ORB) "*provides interoperability between applications on different machines in distributed environments*"[14] and it is a common layer through which objects transparently exchange messages and receive replies. The interfaces that the client objects request and the object implementations provide, are described through the *Interface Definition Language (IDL)*. IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. An interface definition written in IDL specifies completely the interface and each operation's parameters. The IDL concepts are mapped accordingly to the client languages depending on the facilities available in them. This way given an IDL interface, CORBA will generate interface stubs for the client language through which the service can be accessed using the predefined language bindings. In the current status of ORB, language bindings exist only for the C language.

Although the above approaches can provide interoperability support for a large number of applications, they have a number of drawbacks that severely restrict their interoperability support power. The first drawback is the degeneration of the "interface" for which interoperability support is provided to the level of a procedure call. A service is generally provided through an interface that is composed of a set of inter-related procedures. What is of importance is not the actual set of the interface procedures but the overall functionality they provide. By reducing the interoperability "interface" to the level of a procedure call, the inter-relation of the interface procedures is lost, since the interoperability support no longer sees the service interface as a single entity but as isolated procedures. This will create problems in approaches like Polylith's that bridge the differences between the offered and requested service interface, when there is no direct one-to-one correspondence between the interface's procedures (interface mismatch problem).

Interoperability approaches like SLI and CORBA on the other hand do not suffer from the interface mismatch problem, since the client is forced to use a predefined interface. Nevertheless, the enforcement of predefined interfaces (that is, sets of procedures with specified functionality) makes it very difficult to access alternative servers that provide the same service under a different interface. This is an important interoperability restriction since we can neither anticipate nor we can enforce in a open distributed environment the interface through which a service will be provided. With the SLI and CORBA approaches, the service's interface must also be embedded in the client's code. Any change in the server's interface will result in changes in the client code.

Another common characteristic and restriction of the above interoperability approaches is that they require the migration of the procedure parameters from the client's environment to the server's environment. As a result only *migratable* types can be used as procedure parameters. These are the basic data types (integers, strings, reals etc.) and their aggregates (arrays, structures etc.), which we call *data types*. Composite non migratable abstract types, like a database or keyboard type, cannot be passed as procedure parameters. This however is a reasonable restriction since the above approaches focus in interoperability support for systems based on non-object oriented languages where only data types can be defined.

1.2 Object Oriented Interoperability

Although Procedure Oriented Interoperability provides a good basis for interoperability support between non-object oriented language based environments, it is not well suited for a high level interoperability support for environments based on object oriented languages. The reason is that in an object oriented environment we cannot decompose an object in a set of independent operations and data and view them separately, since this will mean loss of the object's semantics. For example, a set of operations that draw a line, a rectangle and print characters on a screen, have a different meaning if they are seen independently or in the context of a window server or a diagram plotting object. In object oriented environments it is the overall functionality of the object that is of importance and not the functionality of the independent operations. That means that the same functionality can be offered with a different interface[1] from different objects found either on the same or in different environments. Interoperability support in an object oriented environment should bridge the interface differences between the various objects offering the same service, while preserving the overall object semantics. We call this kind of interoperability support *Object Oriented Interoperability* (*OOI*).

Object Oriented Interoperability is a generalization of Procedure Oriented Interoperability in the sense that it will use, at its lower levels, the mechanisms and notions of POI. However OOI has several advantages over POI. First of all it allows the interoperation of applications in higher level abstractions, like the objects, and thus supports a more reliable and consistent interoperation. A second advantage is that it supports fast prototyping in application development and experimentation with different object components from different environments. The programmer can develop a prototype by reusing and experimenting with different existing objects in remote (or local) environments without having to change the code of the prototype when the reused object interfaces differ. A last advantage is that since OOI is a generalization of POI, it can be used to provide interoperation between both object oriented and conventional (non-object oriented) environments. Furthermore when OOI support is used for non-object oriented environments it provides a more general frame than POI and can also handle cases where the requested and offered service interfaces do not match.

In this paper we present the concept of Object Oriented Interoperability and describe our prototype implementation. In section 1 we give a brief overview of the previous interoperability approaches and outline the Object Orient Interoperability ideas. In section 2 we present in detail the Object Oriented Interoperability concepts and ideas and in section 3 we describe the prototype implementation. Finally in section 4 we give our conclusions, open issues and further research directions.

2 Overview of Object Oriented Interoperability

We identify two basic components necessary for the support and implementation of object oriented interoperability: *Type Matching* and *Object Mapping*. Type matching provides the means for defining the relations between types on different execution environments based on their functionality abstraction and object mapping provides the run time support for the implementation of the interoperability links.

2.1 Terminology

In the rest of this section we use the term *client interface* to specify the interface through which the client wishes to access a service, and the term *server interface* to specify the actual interface of the server. In addition we will use the term *node* to specify the execution environment of an application (client or server), as for example the Hybrid [4] execution environment or the Smalltalk [2] execution environment. In this sense a node

1. From here on we will use the term "interface" to signify the set of public operations and instance variables, through which functionality and data of an object are accessed.

can span over more than one computer and more than one node can co-exist on the same computer. Although we will assume that the client is in the *local* node and the server in the *remote* node, local and remote nodes can very well be one and the same. With the term *parameter* we mean the operation call parameters *and* the returned values, unless we explicitly state differently. Finally we should note that by the term *user* we mean the person that is responsible for the management of the application.

2.2 Type Matching

In a strongly distributed environment [19] a given service will be offered by many servers under different interfaces. As a result a client wishing to access a specific service from more than one server will have to use a different interface for each server. Although we can develop the client to support different interfaces for the services it access, we might not always be able to anticipate all possible interfaces through which a service can be offered, or force service providers to offer their services via a specific interface. Object Oriented Interoperability approaches this problem by handling all interface transformations, so that a client can use the same interface to access all servers offering the same service. The Type Matching problem consists of defining the bindings and transformations from the interface that the client uses, to the actual interface of the service.

2.2.1 Towards a solution to the Type Matching problem

Ideally we would like to obtain an automatic solution to the Type Matching problem. Unfortunately in the current state of the art this is not possible. The reason is that we have no way of expressing the semantics of the arbitrary functionality of a service or an operation, in a machine understandable form. In practice the best we can do is describe it in a manual page and choose wisely a name so that some indication is given about the functionality of the entity. Nevertheless, since nothing obliges us to choose meaningful names for types, operations or their parameters, we cannot make any assumptions about the meaning of these names. Furthermore even if the names are chosen to be meaningful, their interpretation depends in the context in which they appear. For example a type named *Account* has a totally different meaning and functionality when found in a banking environment and when found in a system administrator's environment. Thus any solution to the Type Matching problem will require, at some point, human intervention since the system can not automatically deduct either which type matches which, or which operation corresponds to which, or even which operation parameter corresponds to which between two matching operations. What the system can do is assist the user in defining the bindings and generate the corresponding implementations.

We distinguish three phases in providing a solution to the Type Matching problem. In the first phase, which we call the *functionality phase*, the user specifies the type or types on the remote environment providing the needed functionality (service). The system can assist the user in browsing the remote type hierarchy and retrieving information describing the functionality of the types. This information can be manual pages, information extracted from the type implementation or even usage examples.

In the second phase, which we call the *interface phase*, the user defines how the operations of the remote type(s) should be combined to emulate the functionality represented by the client's operations. This can a be a very simple task if there is a direct correspondence between requested and offered operations, or a complicated one if the operations from several remote types must be combined in order to achieve the needed result. As in the functionality phase the system can assist the user by providing information regarding the functionality of the operations.

The third phase is the *parameter phase.* After specifying the correspondence between the requested and remote interface operations the user will need to specify the parameters of the remote operations in relation to the ones that will be passed in the local operation call. This might require not only a definition of the correspondence between offered and requested parameters, but also the introduction of adaption functions that will transform or preprocess the parameters. The system can assist the user by identifying the types of the corresponding parameters, reusing any information introduced in the past regarding the relation between types and standard adaption functions, and prompt the user for any additional information that might be required.

2.2.2 Type Relations

In OOI we distinguish three types of type relations, depending on how the local type can be transformed to the remote type. Namely we have *equivalent, translated* and *type matched* types.

Migrating an object from one node to another means moving both of its parts, that is data and operations, to the remote node, while preserving the semantics of the object. However, moving the object operations essentially means that a new object type is introduced on the remote node. This case is presently of no interest to OOI since we wish to support interoperability through the reuse of existing types. Thus in OOI migrating an operation call parameter object means moving the data and using them to initialize an instance of a pre-existing equivalent type. This is most commonly the case with data types, like integers, strings and their aggregates, where the operations exist on all nodes and only the data need to be moved. In OOI when this kind of a relation exists between a type of the local node and a type of the remote node we say that the local type X, has an *equivalent* type X′ on the remote node and we denote it as

X => X′ ; *Local type X has X′ as equivalent type on the remote node.*

Although data types are the best candidates for equivalency relation, they are not the only ones. Other non-data types can also exist for which an equivalent type can be found on a remote node. For example a raster image or a database type can have an equivalent type on a remote node and only the image or database data need to be moved when migrating the object. In general two types can be defined as equivalent if their semantics and structure are equivalent and the transfer of the data of the object are sufficient to allow the migration of their instances. In migrating an object to its equivalent on the remote node, the OOI support must handle the representation differences of the transferred data. In this sense type equivalency of OOI corresponds to representation level interoperability [20].

In an object oriented environment we are more interested in the semantics of an object rather than its structure and internal implementation. For example, consider the Hybrid [13] type string and the CooL [1] type ARRAY OF CHAR. In the general case the semantics of the two types are different: the string is a single object, while the ARRAY OF CHAR is an aggregation of independent objects. Nevertheless when in CooL an ARRAY OF CHAR is used for representing a string, it becomes semantically equivalent and can be transformed to a Hybrid string, although the structure, representation and interfaces of the two types are different. In OOI this type relation is defined as *type translation* and it is noted as

X +> X′ : translationFunction ; *The local type X is translated to type X′ on the remote node, via the defined translation function.*

Translation of the local type to the remote type is done with a user definable translation function. This way the particularities of the semantic equivalency can be handled in case specific way. The user can specify different translations according to the semantics of the objects. For example if the local node is a CooL node and the remote a Hybrid node then we can define two different translations for an ARRAY OF CHAR:

ARRAY OF CHAR +> string : array2string ;
ARRAY OF CHAR +> array of integer : array2array ;

Where in the first case the ARRAY OF CHAR represents a character string, while in the second a collection of characters that need to be treated independently (in Hybrid characters are represented via integers).

Type translation can be compared to Specification Level Interoperability [20], where the interoperability support links the objects according to their specifications. Nevertheless, type translation is more flexible than SLI since it allows multiple translations of the same type according to the specific needs and semantics of the application.

A local type for which bindings to a remote type or types have been defined, as a solution to the Type Match problem, (that is, bindings and transformations from the interface that the client uses, to the actual interface of the service) is said to be *type matched* to the remote node. We can have two kinds of type matched types: multi-type matched and uni-type matched types. Multi type-matched types are the ones that are bound to more that one type on the remote node, and uni-type matched types are the ones that are bound to a single type on the remote node. In OOI we denote that a type X is type matched to types X′ and X′′ on the remote cell as

X –> X′ ; *Local type X is type-matched to remote type X′ .*

X –> < X′, X′′ > ; *Local type X is type-matched to remote types X′ and X′′.*

The target of OOI is to allow access to objects on remote nodes. The basic assumption being that the object in question cannot be migrated on the local node. However, the access and use of the remote object will be done with the exchange of other objects in the form of operation call parameters. The parameter objects can, in their turn, be migrated on the remote node or not. Parameter objects that cannot be migrated on the remote node are accessed on the local node via a type match, becoming themselves servers for objects on the remote node.

Type relations are specific to the node for which they are defined and do not imply that a reverse type relation exists, or that they can be applied for another node. For example, if the local node is a Hybrid node and the remote is a C++ node, the Hybrid type boolean has as equivalent in the C++ node an int (integer) (booleans in C++ are presented by integers), while the reverse is, in general, false.

2.2.3 To Type-Match or not to Type-Match?

Type matching is a general mechanism for interoperability support and it can be used in all cases in place of equivalency and translation of types. However, the existence of translation and equivalency of types is needed for performance reasons since accessing objects through the node boundary is an expensive operation. If an object is to be accessed frequently on the remote node, then it might be preferable to migrate it, either as equivalent or translated type. For example, it is preferable to migrate "small" objects, like the data types, rather than access them locally. Nevertheless the user has always the possibility to access any object locally, even an integer if this is needed, as it might the case with an integer that is stored at a specific memory address which is hardwired to an external sensor (like a thermometer) and which continuously updated. This can be done by defining a type match and using it in the parameter's binding definitions.

A typical scenario we envisage in the development of an application with OOI support is the following. The user (application programmer) will first define a set of type matchings for accessing objects on remote nodes. These will be used in the development of the application prototype. When the prototype is completed the user will measure the performance of the prototype and choose for which types a local implementation is to be provided. For these types an equivalency of translation relation will also be established, possibly on both nodes, so that they can be migrated and accessed locally. This way the performance of the prototype will be improved. This process can be repeated iteratively until the performance gains are no longer justifiable by the implementation effort.

One of the major advantages of the OOI approach is that in the above scenario the application prototype will not be modified when local implementations of types are introduced[1] and the type relations change. The new type relations are introduced in the OOI support and do not affect the application programs.

2.3 Object Mapping

Whereas type matching maintains the static information of the interoperability templates, object mapping provides the dynamic support and implementation of the interoperability links. We distinguish two parts in object mapping: the static and the dynamic. The static part of object mapping is responsible for the creation of the classes that implement the interoperability links as specified by the corresponding type matching. The dynamic part on the other hand, is responsible for the instantiation and management of the objects used during the interoperation.

2.3.1 Inter-Classes and Inter-Objects

The essence of object mapping is to dynamically introduce in the local node the services of servers found on other nodes. This however must be done in such way so that the access of the services is done according to the local conventions and paradigms. In an object oriented node this will be achieved with the instantiation of a local object that represents the remote server, which in OOI we call an *inter-object*. An inter-object differs

1. With the exception of a possible recompilation if dynamic linking is not supported.

from a proxy, as defined in [18], in three important points. First in contrast with a proxy, an inter-object and its server can belong to different programming and execution environments and thus they follow different paradigms, access mechanisms and interfaces. The second difference is that while a proxy provides the only access point to the actual server, that is the server can be accessed *only* via its proxies, this is not the case with inter-objects. Objects on the same node with the server can access it directly. An inter-object provides simply the gateway for accessing the server from remote nodes. Finally while a proxy is bound to a specific server, an inter-object can dynamically change its server or even access more than one server combining their services to appear as a single service on the local node.

An inter-object is an instance of a type for which a type match has been defined. The class (that is, the implementation of a type) of the inter-object is created by the object mapper from the type match information and we call it *inter-class*. An inter-class is generated automatically by the object mapper and it includes all code needed for implementing the links to the remote server or servers.

2.3.2 Dynamic Support of the Object Mapping

After the instantiation of an inter-object and the establishment of the links to the remote server, the controlling application will start calling the operations of the inter-object passing other objects as parameters. OOI allows objects of any type to be used as a parameters at operation calls. The object mapper will handle the parameter objects according to their type relations with the remote node. This way objects whose type has an equivalent or translated one on the remote node, will be migrated, while objects for which a type match exists will be accessed through an inter-object on the remote node.

In the case where no type relation exists for the type of a parameter object, the object mapper will invoke the type matcher and ask the user to provide a type relation. This way type relations can be specified efficiently taking into account the exact needs and circumstances of their use. In addition the dynamic definition of type relations during run time relieves the user from the task of searching the type hierarchy for undefined type relations. Also the incremental development and testing of a prototype becomes easier since no type relations need to be defined for the parts of the prototype that are not currently tested.

3 Prototype Implementation

A prototype implementation of Object Oriented Interoperability support was designed and developed for the Hybrid cell [5]. In this section we present the prototype implementation and discuss the related issues. In our presentation we are using interoperability examples for Hybrid and CooL. Hybrid is an object oriented language designed [13][7] and implemented [4] at the University of Geneva, whereas CooL is a an object oriented language designed and implemented in the ITHACA Esprit [1] project and which is now a product from Siemens-Nixdorf Inf. AG.

The implementation of the OOI support was part of the Cell prototype implementation [8]. The Cell is a frame for the design of strongly distributed object based systems [6]. In the Cell frame each node is transformed to a *cell* composed by a *nucleus* and a *membrane*. The nucleus is the original execution environment while the membrane surrounds the nucleus and is responsible for all external (to the nucleus) communication. The main goal of the Cell frame is to allow the objects of a node (nucleus) to transparently access and use services found on other heterogenous nodes, without introducing any changes to the nucleus.

3.1 Type Matching on the Hybrid cell

In order to provide support for type matching in the Hybrid cell, we designed and implemented a *Type Matching Specification Language (TMSL)* that allows the user to express type relations in a syntax very similar to the Hybrid language syntax. Although the design of the Hybrid TMSL (which we will refer as *H-TMSL*) has been influenced by the interface language NIMBLE [16], H-TMSL is more general since it allows any type to be used as parameter of an operation. In the following we describe the H-TMSL using examples of how the type relations for a CooL cell can be expressed. The full grammar of the H-TMSL is given in Annex I.

In the Hybrid cell the type relations defined in H-TMSL are stored in membrane objects. Nevertheless, in order to allow easier access from the UNIX environment (for debugging and verification) the definitions are also stored and kept synchronized in a UNIX file in H-TMSL syntax. At initialization of the membrane the file containing the type relation definitions is opened and all information is loaded into the membrane objects.

3.1.1 Type Relations

A type relation in H-TMSL is defined for a specific remote cell which is identified by its name.[1] For the examples given bellow we assume that the local Hybrid cell is named HybridCell and the remote CooL cell is named CooLCell. The general syntax of a type relation on the Hybrid cell is

```
IdOfRemoteCell :: <TypeRelation> ;
```

where TypeRelation can be either equivalent, translated or type matched and IdOfRemoteCell is the id of the remote cell, which in the case of the CooL cell is CooLCell.

Equivalent and Translated types.

In both CooL and Hybrid integers and booleans are equivalent types. On the Hybrid cell this is expressed as

```
CooLCell :: integer => INT ;
CooLCell :: boolean  =>  BOOL ;
```

Although the notion of a *string* exist in both languages, in CooL strings are represented as arrays of characters while in Hybrid they are *basic data types*. Thus the relation between them is of a translated type

```
CooLCell :: string +> ARRAY OF CHAR : string2arrayOfChar ;
```

In the CooL cell the corresponding definitions will be:

```
HybridCell :: INT => integer ;
```

1. The names of the cells are managed by the membranes and are specific to it.

```
HybridCell :: BOOL => boolean ;
HybridCell :: ARRAY OF CHAR +> string : arrayOfChar2string ;
```

In the definition of translated types we specify a translation function, like string2array-OfChar and arrayOfChar2string, which performs the data translation.

Because type equivalency and translation imply knowledge and ability to access the internal representation of the objects, we do not allow, in the present implementation, the dynamic introduction of equivalent and translated types by the user. All information about equivalent and translated types is defined statically and loaded at initialization.

Type Matched types.

In contrast to equivalent and translated types, type matchings can be defined dynamically at run time by the user. A type can be matched to either a single remote type or to a collection of remote types (*multi-type match*). For example if we have on the local Hybrid cell a type windowServer, which is matched to the type WINDOW_CONTROL of the remote cell, the type match will be expressed as

```
CooLCell :: windowServer -> WINDOW_CONTROL  {<operation bindings>} ;
```

while a multi-type match will be expressed as

```
CooLCell :: windowManager -> < WINDOW_CONTROL, SCREEN_MANAGER >
                  { <operation bindings>} ;
```

When an object of the local nucleus in its attempt to access a service creates an instance of a type matched type (an inter-object), a corresponding instance of its type matched type will be created on the remote cell. However, there are cases where we do not want a new instance to be created on the remote cell but we need to connect to an existing server. This for example can be the case with a data-base object. We do not want an instance of an empty data-base but we want to use the existing one with all its stored data. In H-TMSL this is noted with the addition of @ at the of remote type name:

```
CooLCell :: personnel ->  PERMANENT_PERSONEL_DB @
                      { <operation bindings>} ;
```

If there are more than one instances of type PERMANENT_PERSONEL_DB at the CooL cell then it is up to the membrane to choose which one will be used. However in the case of an instantiation of an inter-object due to a parameter mapping, the object mapper will always bind the inter-object to the corresponding (pre-existing) parameter object.

In the rest of this section we describe the H-TMSL type matching syntax using as examples a Hybrid type windowServer, which defines in the Hybrid cell the interface through which a window server is to be accessed (requested interface), and a CooL type WINDOW_CONTROL which provides an implementation of the a window server (offered interface). For simplicity we assume that the operation names of the two types describe accurately the functionality of the operations. For example the operation named newWindow creates a new window, while the operation get_Position returns the position pointed by the pointing devices (i.e. mouse, touch-screen etc.).

The Hybrid type windowServer (Figure 1) has five operations. Operations newWindow and newSquareWin return the id of the newly created window or zero in case of failure. Operation refreshDisplay returns true or false signifying success or failure. Operation readCoordinates returns the coordinates of the active point on the screen as read

```
type windowServer : abstract {
      newWindow : (integer #{ : topLeftX #}, integer #{ : topLeftY #},
                        integer #{ : botRightX #}, integer #{ : botRightY #})
                    -> integer #{: windowId #} ;
      newSquareWin : (integer #{ : topLeftX #}, integer #{ : topLeftY #},
                          integer #{ : side #} ) -> integer #{ : windowId #} ;
      refreshDisplay : (display ) -> boolean ;
      readCoordinates : ( mouse, keyboard, touchScreen, integer #{ : scaleFactor #} )
                           -> point ;
      windowSelected : (mouse, keyboard, touchScreen ) -> integer ;
} ;
```

Figure 1 Hybrid Type windowServer.

from the pointing devices and operation windowSelected returns the id of the currently selected window or zero if no window is selected.

The CooL type WINDOW_CONTROL (Figure 2) has 4 methods. The methods cre-

```
TYPE WINDOW_CONTROL =
     OBJECT
          METHOD create_win ( IN botRightX : INT, IN botRightY : INT,
                              IN topLeftX : INT, IN topLeftY : INT, IN color : INT ) : INT
          METHOD redisplay_all (IN  display : DISPLAY) : INT
          METHOD get_Position (IN inDevices : IO_DEVICES, IN scaling : INT)
                        : POSITION
          METHOD select_Window (IN position : POSITION) : INT
     BODY
     ...
     END OBJECT
```

Figure 2 CooL Type WINDOW_CONTROL

ate_win and select_Window return the id of the newly created window and of the window into which the specific position is found, or -1 in case of an error. Method redisplay_all returns 0 or 1 signifying failure or success, and method get_Position returns the position pointed by the I/O devices (i.e. keyboard, mouse, touch-screen etc.) as adapted by the scaling factor.

3.2 Binding of Operations

Although type WINDOW_CONTROL provides all the functionality that type windowServer requires, this is done via an interface different to the one that windowServer expects. In general in H-TMSL we anticipate two levels of interface differences. First in the required parameters (order, type etc.) and second in the set of supported operations, that is, different number of operations with aggregated, segregated or slightly[1]

1. The term is used loosely and it is up to the user to define what constitutes a minor (slight) difference in functionality.

different functionality. The resolution of these differences corresponds to the parameter and interface phases of the type matching definition.

3.2.1 Parameters phase

Assuming that the functionality of the provided operation corresponds to the requested functionality, the differences between the parameters passed to the local operation call (offered parameters) and of the parameters required by the remote operation (requested parameters) can fall into one or more of the following categories:

- Different order of parameters. For example the first parameter of the local operation might correspond to the second on the remote operation.
- Different representation of the information held by the parameter. For example a boolean condition TRUE or FALSE can be represented locally by an integer while on the remote operation the string "TRUE" or "FALSE" might be expected.
- Different semantic representation of the information. For example if we have a Hybrid array with ten elements indexed from 10 to 19, an equivalent array in CooL will be indexed 1 to 10. Thus an index, say 15, of the Hybrid array should be communicated as 6 to the CooL cell.
- Different number of parameters. The requested parameters might be more or less than the offered ones. In this case the parameters offered might include all information needed or more information might be required.

H-TMSL anticipates all the above differences and allows the user to specify the needed transformations for handling them.

Migrated parameters

In our example we consider first the operations newWindow and create_win which have the same functionality specification. The binding of newWindow to create_win is expressed in H-TMSL as

```
newWindow : create_win(3, 4, 1, 2, int17() ) ^ RET ;
```

Operation newWindow offers four parameters which are identified by their position with a positive integer (1 to 4). Method create_win will be called with these parameters transposed. Its first parameter will be the third passed by newWindow, the second will be the fourth and so on. The fifth parameter of create_win specifies the color of the new window. This information does not exists in the offered parameters. Nevertheless, in this case, we can use a default value with the use of an *adaption function*, like int17(). (Adaption functions are described in the next paragraphs.) The returned value from create_win, noted as RET in H-TMSL, is passed back to the Hybrid cell and becomes the value that newWindow will return.

In the above operation binding definition we assume that a relation for the CooL and Hybrid integers exists. That is we assume that on the Hybrid cell we have

```
CooLCell :: integer => INT ;
```

and on the CooL cell

```
HybridCell :: INT => integer ;
```

This way the migration of the parameters and returned values will be handled automatically.

Operation newSquareWin does not exist in the interface of WINDOW_CONTROL but its functionality can be achieved by operation create_win called with specific parameter values. That is we can have

```
newSquareWin : create_win (bottomRX(1, 3), bottomRY(2, 3),1, 2, int17()) ^ RET;
```

where functions bottomRX and bottomRY are adaption functions. Adaption functions are user defined functions, private to the specific type match. They provide the means through which the user can adapt the offered parameters to a format compatible to the requested parameters. They can be called with or without parameters. The parameters to be passed to the adaption functions can be any of the offered parameters or even the result of another adaption function. In the type matching definition of H-TMSL the adaption functions are included at the end of the type match definition between @{ and @}. Thus for the previous example we have the following adaption functions:

```
@{
        bottomRX : (integer : topLeftX, side ) -> integer ;
                { return (topLeftX + side ) ; }

        bottomRY : (integer : topLeftY, side ) -> integer ;
                { return (topLeftX - side ) ; }

        int17 :  -> integer ;
                { return (17) ; }
}@
```

The adaption functions will be invoked locally (that is, in our example, in the Hybrid cell) and their result will be passed as parameter to the remote call (create_win). An adaption function is effectively a private operation of the inter-class and as such it can access its instance variables or other operations.

Mapped Parameters

When the parameter cannot be migrated to the remote cell, that is when there is no corresponding equivalent or translated type, it should be accessed on the local cell. This will be done via a *mapping* of a remote object to the local parameter according to an existing type match. In our example this will need to be done for the refreshDisplay operation and redisplay_all method.

The parameter passed to refreshDisplay is an object of type display which cannot be migrated to the CooL cell. Thus it must be accessed on the Hybrid cell via a mapping on the CooL cell. For this a type match must exist on the CooL cell to the Hybrid display type.

```
HybridCell :: DISPLAY -> display { .... } ;
```

This way the binding of refreshDisplay to redisplay_all is expressed as

```
refreshDisplay : redisplay_all ( 1 : display <- DISPLAY ) ^ int2bool(RET) ;
```

meaning that the first parameter of the method redisplay_all will be an object mapped to the first parameter passed to the operation refreshDisplay, according to the specified type match on the CooL cell. In addition the returned value of redisplay_all, which is an integer, is transformed to a boolean via the adaption function int2bool which is defined as following:

```
@{
        int2bool : ( integer : intval ) -> boolean ;
                {
                        if ( intval ?= 0 )   { return (FALSE) ; }
                        else                 { return ( TRUE ); }
                }
@}
```

Multi-type mapped parameters

In H-TMSL we also anticipate the case where the functionality of a type is expressed by the composite functionality of more than one type on the remote cell. In our example this is the case for the CooL type IO_DEVICES, which corresponds to the composite functionality of the Hybrid types mouse, keyboard and touchScreen.

```
HybridCell :: IO_DEVICES -> < keyboard @, mouse @, touchScreen @ > { ... } ;
```

Note that in this example the IO_DEVICES inter-object will be connected to the existing keyboard, mouse and touchScreen objects on the Hybrid cell.

The definition of multi-type match operation bindings is similar with that of single type match bindings, but with the definition of the operation's type. If for example we assume that type IO_DEVICES has a method read_keyboard which corresponds to the operation readInput of the Hybrid keyboard type, the binding would be expressed as

```
read_keyboard : keyboard.readInput (...) ^ ... ;
```

In fact this syntax is the general syntax for the definition of an operation binding and can be used in both single or multi type matchings. Nevertheless for simplicity in single type matchings the definition of the corresponding type can be omitted since there is only one type involved.

In our original example, the binding of the Hybrid operation readCoordinates to the operation get_Position will be expressed as

```
readCoordinates : get_Position ( < 2,1, 3 >  : < keyboard, mouse, touchScreen >
                                  <- IO_DEVICES,  4 )  ^ RET
```

assuming that we have on the CooL cell the relation

```
HybridCell :: POSITION +> point ;
```

3.2.2 Interface adaption

When defining the operation bindings between two types from different environments there will be cases where the functionality of the local operation is an aggregation of the functionality of more than one remote operations. Adapting a requested operations interface to an offered one might require anything from simple combinations of the operations up to extensive programming. In order to simplify the user's task, H-TMSL allows the definition of simple operation combinations in the type match specification. For example the functionality of the Hybrid operation windowSelected can be obtained with the combination of the CooL methods get_Position and select_Window. The operation binding is thus:

```
windowSelected : select_Window (  WINDOW_CONTROL.get_Position (
      < 2,1, 3 >  : < keyboard, mouse, touchScreen >  <- IO_DEVICES, 4 ) )  ^ RET ;
```

This defines that the method get_Position will first be called on the remote CooL cell and its result will not be returned to the calling Hybrid cell but it will be used as the first parameter to the select_Window method. Since the result of the get_Position method is not returned to the Hybrid cell, there is no need for a type relation of the CooL type POSITION to exist on the Hybrid cell.

To be noted that the remote operation call is defined as WINDOW_CONTROL.get_-Position, that is *with* the type that it belongs to, so that it be can distinguished from the adaption functions.

3.3 Object Mapping

Once a complete type match definition has been specified, the information is passed to the object mapping service (object mapper) which is responsible for the run-time support of the OOI. The object mapper will generate dynamically an inter-class for the local matched type, which in our example is the windowServer, and add it into the Hybrid workspace. The operations of the inter-class are generated from a general template and their task is to forward the operation call to the remote server [8]. When an instance of the type is requested, an inter-object will be instantiated connected to the remote server object, that is the instance of the WINDOW_CONTROL, with its operation bound to the corresponding server's operations. The object mapping service is responsible for locating the target cell and establishing and maintaining the communication channel(s). We then say that the instance of the WINDOW_CONTROL in the CooL cell is mapped into the Hybrid cell (Figure 3).

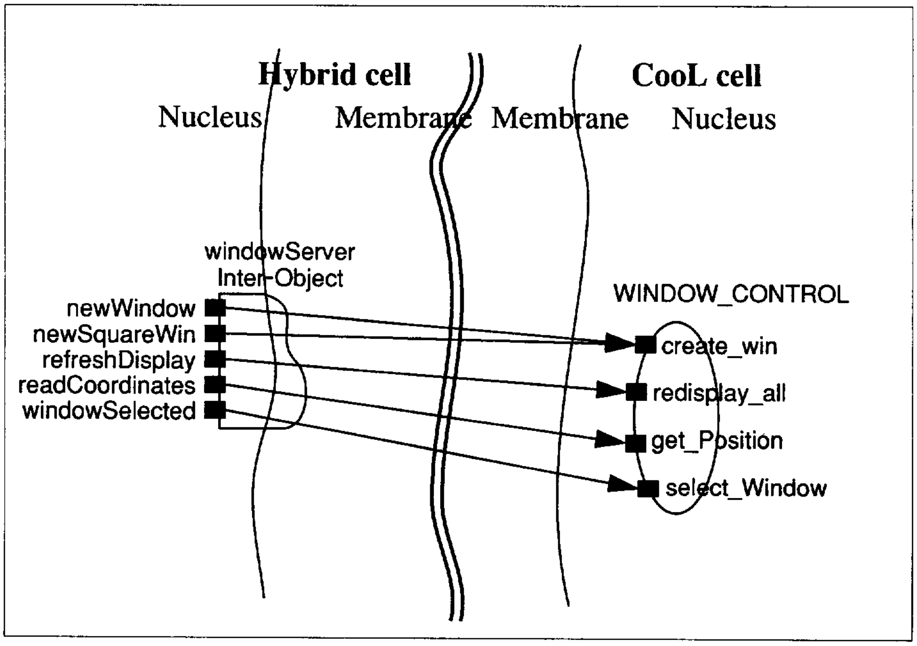

Figure 3 Object Mapping.

In order to outline the functionality of the object mapping service we will describe the actions taken when an operation of the windowServer inter-object is called. For our example we consider the operation readCoordinates, which is called with four parameters: a keyboard object, a mouse object, a touchScreen object and an integer (Figure 4)

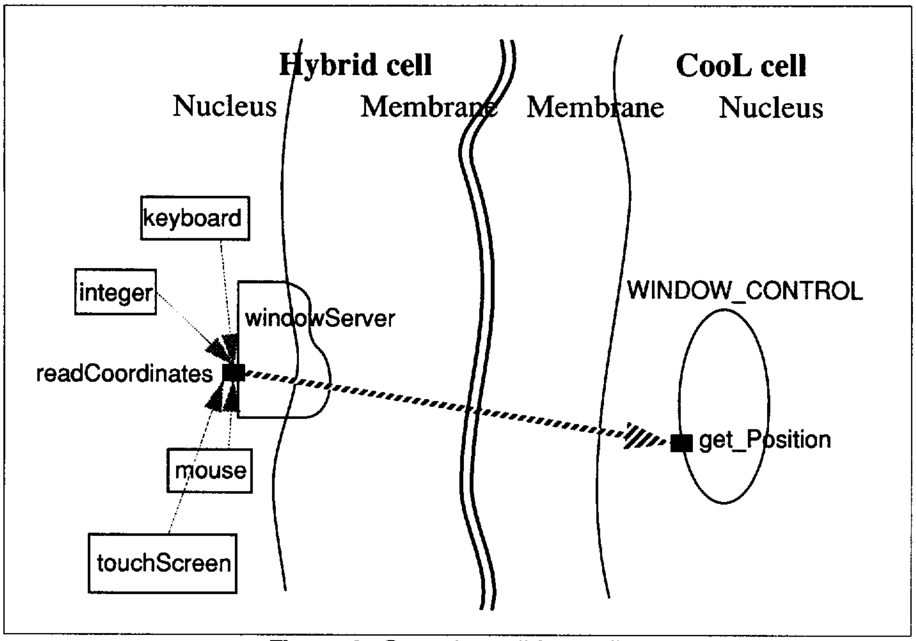

Figure 4 Operation call forwarding

and which is bound to the method get_Position.

readCoordinates : get_Position (< 2,1, 3 > : < keyboard, mouse, touchScreen > <- IO_DEVICES, 4) ^ RET

From the 4 parameters passed to operation readCoordinates, the first three ones (keyboard, mouse and touchScreen) cannot be migrated to the CooL cell but must be accessed locally via a multi-type match of the CooL type IO_DEVICES. The fourth parameter is an integer for which an equivalent type exists on the CooL cell and thus it can be migrated to it. This information is known to the object mapper since it is included in the type match specifications. The local object mapper will contact the remote object mapper and request the instantiation of two object: an inter-object of type IO_DEVICES connected to the Hybrid objects keyboard, mouse and touchScreen and an INT object initialized to the value of the integer parameter (Figure 5).

When the transfer of the parameters has been completed the object mapper will proceed in the invocation of the remote operation. The remote object mapper will be instructed to call the operation get_Position passing it as parameters the IO_DEVICES inter-object and the INT object (Figure 6). The CooL object mapper will invoke the method and receive the result, an object of type POSITION. Because for the CooL type POSITION there is a translation to the Hybrid type point, the CooL object mapper will instruct the Hybrid object mapper to instantiate an object of type point which it will in-

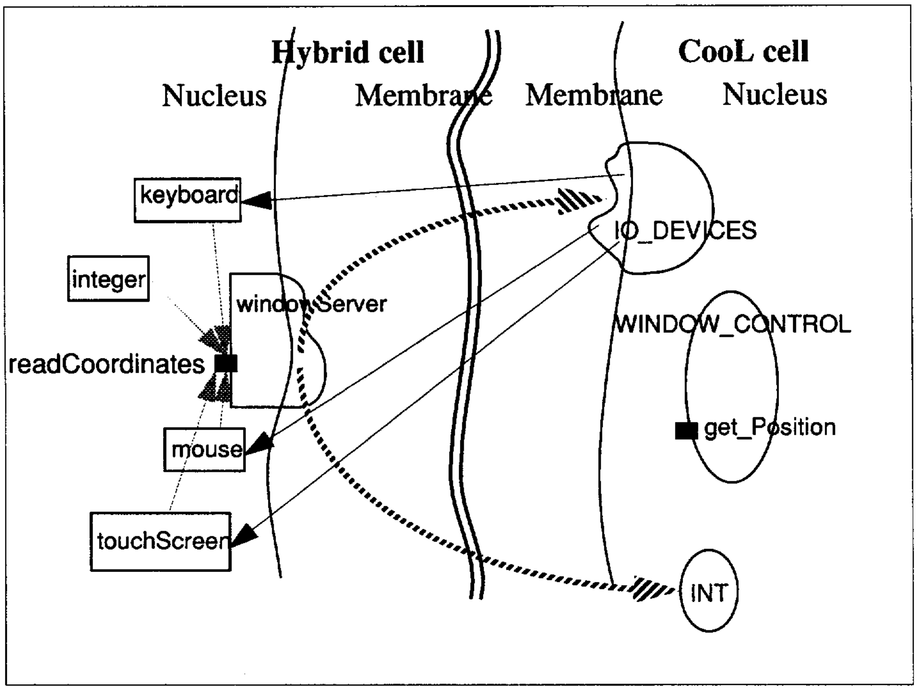

Figure 5 Parameters' transfer

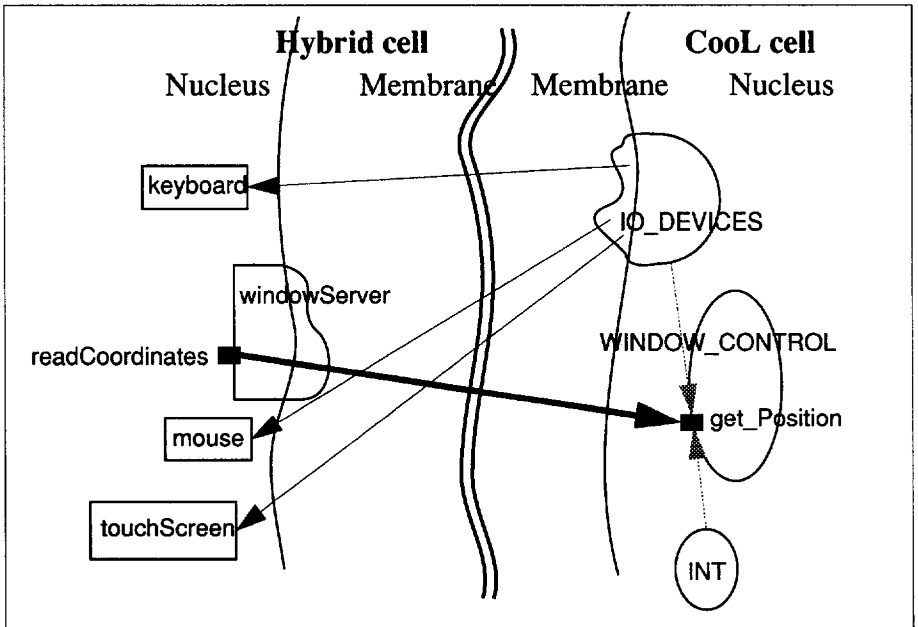

Figure 6 Remote operation invocation

itialize to the translated value of the POSITION object. The Hybrid object mapper will then be instructed to receive the result of the readCoordinates operation call in the instance of point. This will finally be the result returned to the caller of the readCoordinates operation.

During the transfer of parameters the object mapper might encounter a type for which no type relation has been defined. For example it might be that on the CooL cell there is no type relation for the type IO_DEVICES. This is possible since the type matcher of the Hybrid cell does not verify during a type match definition the existence of reverse type relations (i.e. from the remote cell to the local). In this case when the Hybrid object mapper will request the instantiation of an IO_DEVICES inter-object, the CooL type matcher will invoke dynamically the type matcher requesting the definition of the type match. The user will then be required to define on the fly a type match for the IO_DEVICES type. Once this is done the object mapper will resume the transfer of the parameters. This way an application can be started even without any type relations defined. The object mapper will prompt the user to define all needed type relation during the first run of the application.

4 Conclusions and issues to be studied

We have presented the concept of Object Oriented Interoperability and our prototype implementation that demonstrates both the feasibility of the ideas and the implementation related issues. Object Oriented Interoperability is an extension and generalization of the Procedure Oriented Interoperability approaches taken in the past. While Procedure Oriented Interoperability approaches support interoperability at the operation call level, linking applications through independent operation call, Object Oriented Interoperability (OOI) supports interoperability at the object level. That is, it considers the object, an inseparable set of operations and data, as the basic interoperation unit. OOI provides interoperation support based on the abstract functionality of the objects without being restricted by the specific interface through which their functionality is offered.

Object Oriented Interoperability offers many advantages over traditional Procedure Oriented Interoperability approaches. In contrast to Procedure Oriented Interoperability approaches which require parameter objects to be migrated to the remote environment, OOI allows parameter objects to access locally if they cannot be migrated. As a result OOI places no restriction to the interfaces used in interoperation. A second advantage is that OOI does not require exactly matching interfaces nor does it force the interface through which a service must be accessed. The application designer can decide on the interface that he wants to use for accessing a service and use it for accessing not only the target server but also alternative servers offering the same service under different interfaces. Another advantage of OOI is that it makes no assumptions about the existence and semantics of types in the interoperating environments. Each type, even the simplest and most banal integer type, must be explicitly related to a type on the remote environment. This way OOI provides flexibility in the interconnection of diverse environments based on different models and abstractions.

One of the “disadvantages” of OOI comes from the fact that it does not enforce a common global representation model (Type Matching Specification Language) for expressing the interoperability bindings. Each execution environment is free to choose its own language. As a result the interoperability type matching specifications for a server need to be defined independently by the user for each execution environment. However, bilateral mappings can offer a higher flexibility when the interoperating languages support special features. For example, a common interface definition language, like the CORBA IDL, does not include the notion of a *transaction*; thus, even when the interoperating languages support transactions, like Argus [11] and KAROS [2], their IDL based interoperation will not be able to use transactions.

Our prototype implementation for OOI support allowed us to identify several issues that need further studying. A first issue concerns the security aspects of OOI support. That is, not only the authorization and verification of the interoperating entities but also, and most important, the implications of allowing a remote user to define new type relations in the local environment and then instantiate the resulting inter-classes. The questions in this issue are who has the right to create new type relations and how this can be controlled. A second security aspect concerns the inter-relations with third parties. A user can define a type match and establish links for a type of a second cell that is itself type-matched to a third cell. In this case we access the third cell indirectly via the second cell, using accesses privileges that we might not have locally.

A second issue that needs to be studied concerns the public instance variables in the interconnection of application developed in different programming languages. The issue here is dual: first whether the public instance variables of a local type are meaningful to an inter-object and if so how they are bound and used, and second how do we handle the public instance variables of two applications written in different languages where one supports public instance variables while the other does not.

A last issue concerns the refinement of the TMSL syntax. For example, the choice of using numbers for expressing the parameter correspondence was not an optimal one. A better choice would have been to use a code, like #1, #2 etc., so that it would be easier to use integers for parameter extensions and not adaption functions. The syntax refinement will also allow us to define a TMSL framework that can adaptedable to more than one language.

We plan to continue our research by refining the Object Oriented Interoperability specifications and studying the open issues presented. In parallel we plan to design and develop OOI support prototypes for different executions environments, based on both Object Oriented and non-Object Oriented languages. Once the OOI support prototypes are developed we will use them for interconnecting different applications, both existing and specially developed, and study the related issues and problems.

References

[1] Denise Bermek and Hugo Pickardt, "HooDS 0.3/00 Pilot Release Information", ITHACA.SNI.91.D2#4, August 28, 1991, Deliverable of the ESPRIT Project ITHACA (2705).

[2] Rachid Guerraoui, *"Programmation Repartie par Objets: Etudes et Propositions"* Ph.D. Thesis, Universite de Paris-Sud, October 1992.

[3] Adele Goldberg "Smalltalk-80", Addison Wesley 1984.

[4] Yoshinori Kishimoto, Nobuto Kotaka, Shinichi Honiden, "OMEGA: A Dynamic Method Adaption Model for Object Migration", Laboratory for New Software Architectures, IPA Japan (Working Paper)

[5] D. Konstantas, O.M. Nierstrasz and M. Papathomas, "An Implementation of Hybrid," in *Active Object Environments*, ed. D. Tsichritzis, pp. 61-105, Centre Universitaire d'Informatique, University of Geneva, 1988.

[6] Dimitri Konstantas, "Cell: A model for Strongly Distributed Object Based Systems", in *Object Composition* ed. D. Tsichritzis, CUI, University of Geneva, 1991. Also presented in the ECOOP 91 Workshop "*Object-Based Concurrent Computing*".

[7] Dimitri Konstantas, "Design Issues of a Strongly Distributed Object Based System," Proceedings of 2nd International Workshop for Object-Orientation in Operating Systems (I-WOOOS '91), IEEE, Palo-Alto, October 17-18, 1991.

[8] Dimitri Konstantas, "Hybrid Update" in *Object Frameworks*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1992.

[9] Dimitri Konstantas , "Hybrid Cell: An Implementaiton of an Object Based Strongly Distributed System", Proceedings of the *International Symposium on Autonoums Decentralized Systems ISADS 93*, Kawasaki, Japan, March 30, 1993. Also in *Object Frameworks*, ed. D. Tsichritzis, CUI, University of Geneva, 1992 under the title "The Implementation of the Hybrid Cell".

[10] Jintae Lee and Thomas W. Malone, "How can Groups Communicate when they use Different Languages? Translating Between Partially Shared Type Hierarchies", Proceeding of the *Conference on Office Information Systems*, March 1988, Palo Alto CA.

[11] Barbara Liskov, Dorothy Curtis, Paul Johnson and Robert Scheifler, "Implementation of Argus," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 111–122, ACM, Austin TX (USA), November 1987.

[12] Oscar M. Nierstrasz, "Active Objects in Hybrid", in *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, N. Meyrowitz (ed.), Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987 243-253

[13] *The Common Object Request Brocker: Architecture and Specification*, Object Management Group and X Open, Document Number 91.12.1 Revision 1.1

[14] Xavier Pintado, "Gluons: Connecting Software Components", in *Object Composition* Technical report ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1991.

[15] James M. Purtilo and Joanne A. Atlee, "Module Reuse by Interface Adaption", *Software Practice & Experience*, Vol. 21 No 6, June 1991.

[16] Ken Sakamura, "Programmable Interface Design in HFDS", Proceedings of the Seventh TRON Project Symposium, Springer-Verlag 1990, Tokyo, 1990.

[17] Marc Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle," 6th International Conference on Distributed Computing Systems, Boston, Massachusetts, May 1986.

[18] Peter Wegner, "Concepts and Paradigms for Object Oriented Programming.," ACM OOPS Messenger, vol. 1, no. 1, August 1990.

[19] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt and Peri L. Tarr, "Specification Level Interoperability," Communications of ACM, vol. 34, no. 5, May 1991.

Annex I: Type Matching Programming Language.

```
typeMatchDef      : remoteCellId ‘::’ typeMatch ‘;’
typeMatch         : localType ‘->’ remoteTypes typeMatchSpec
                  | localType ‘=>’ remoteType [ ‘:’ transFunction ]
                  | localType ‘+>’ remoteType [ ‘:’ transFunction ]
remoteTypes       : ‘<’ remoteTypeList ‘>’
remoteTypeList    : remoteType [‘@’] [‘,’ remoteTypeList]
typeMatchSpec     : ‘{’ operMatchList ‘}’ [ adaptDefList ]
adaptDefList      : ‘@{’ Program ‘}@’ [adaptDefList]
operMatchList     : operMatch [operMatchList]
operMatch         : localOpName ‘:’ remoteOpDef ‘(’argMatchList ‘)’ ‘^’ returnValDef ‘;’
remoteOpDef       : remoteType ‘.’ remoteOpName
argMatchList      : argMatch [‘,’ argMatchList]
argMatch          : localArgId
                  | adaptFunct ‘(’ localArgId ‘)’
                  | localArgId ‘:’ localType ‘<-’ remoteType
                  | ‘<’ localArgIdList ‘>’ ‘:’ ‘<’ localTypeList ‘>’ ‘<-’ remoteType
                  | remoteOpDef ‘(’ argMatchList ‘)’
returnValDef      : RET
                  | adaptFunct ‘(’ RET ‘)’
                  | RET ‘:’ localType ‘->’ remoteType
localArgIdList    : localArgId [‘,’ localArgIdList]
localTypeList     : localType [ ‘,’ localTypeList]
localArgId        : SMALL_INTEGER
localType         : STRING
remoteType        : STRING
remoteOpName      : STRING
remoteCellId      : STRING
transFunction     : STRING
adaptFunct        : STRING
Program           : Program code in Hybrid.
```

Annex II: Type Match definition example

```
CooLCell :: windowServer -> WINDOW_CONTROL {
        newWindow : create_win(3, 4, 1, 2, int17() ) ^ RET ;
        newSquareWin : create_win ( bottomRX(1, 3), bottomRY(2, 3),1, 2, int17() )
                        ^ RET ;
        refreshDisplay : redisplay_all ( 1 : display <- DISPLAY ) ^ int2bool(RET) ;
        readCoordinates : get_Position
                        (< 2,1, 3 > : < keyboard, mouse, touchScreen > <- IO_DEVICES, 4 )
                        ^ RET
        windowSelected : select_Window (
                        WINDOW_CONTROL.get_Position
                        ( < 2,1, 3 >  : < keyboard, mouse, touchScreen > <- IO_DEVICES,
int1() )
                        )  ^ RET ;
}
@{
        bottomRX : (integer : topLeftX, side ) -> integer ;
                { return (topLeftX + side ) ; }

        bottomRY : (integer : topLeftY, side ) -> integer ;
                { return (topLeftX - side ) ; }

        int17 :  -> integer ;
                { return (17) ; }

        int1 :  -> integer ;
                { return (1) ; }

        int2bool : ( integer : intval ) -> boolean ;
                {
                        var boolval : boolean ;
                        if ( intval ?= 0)       { boolval := FALSE ; }
                        else                    { boolval := TRUE ; }
                        return (boolval) ;
                }
@} ;
```