Chapter 3

Interoperation of Object-Oriented Applications

Dimitri Konstantas

Abstract One of the important advantages of the object-oriented design and development methodology is the ability to reuse existing software modules. However the introduction of many programming languages with different syntax, semantics and/or paradigms has created the need for a consistent inter-language interoperability support framework. We present a brief overview of the most characteristic interoperability support methods and frameworks allowing the access and reuse of objects from different programming environments and focus on the interface bridging object-oriented interoperability support approach.

3.1 Reusing Objects from Different Environments

One of the problems that people face when travelling from one country to another concerns the operation of electric appliances, like electric razors and coffee machines. A person living in Switzerland, for example, travelling to Germany will not be able to "plug in" and use his coffee machine as he is used in doing when back home. The reason is simply that the "interfaces" for connecting to the electricity distribution network, that is the plug of the appliance and the wall socket, are different. Our traveller will need to employ a small inexpensive adaptor in order to bridge the differences of the "interfaces". But things are not always that simple. If the same person is travelling to North America he will discover that not only is his (Swiss) plug different from the (North American) wall socket, but also that the electricity voltage differs. Fortunately also in this case a simple solution exists: the use of a transformer that will convert the North American voltage (110 V) to the Swiss standard (220 V).

Dimitri Konstantas, "Interoperation of Object-Oriented Applications," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 69-95, Prentice Hall, 1995.

In object-oriented programming where the reuse of objects is highly encouraged we face similar problems when we wish to access or reuse objects that are programmed in different programming languages. A programmer implementing an application in C++ cannot easily (re)use (if at all) objects and code written in Smalltalk [5] or even replace, without resorting to extensive reprogramming, a C++ object with some other one performing the same function but under a different interface. What we need are concepts similar to the electricity transformer and plug adaptor that will allow us to bridge the differences between the interfaces and paradigms of objects programmed in different languages.

In general we can classify the problems of bridging the differences between objects into three categories. The first category includes the computation differences between the objects, like the low-level data representations; the second category includes the syntactic particularities of the object interfaces, like the operation names and the required parameters; the third category includes the differences of the semantic and functional behaviour of the objects, like the representation of a collection of objects as an array or as a linked list. We will refer to the bridging of all these differences for the reuse and access of objects written in one or more languages as the *interoperability support* problem.

Interoperability 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 interoperability support between entities is the mutual exchange of information and the use of resources available in other environments.

During the past few years several approaches have been taken for the introduction of interoperability support. We classify these approaches in two ways. First depending on the way that they solve the interface differences' problem and second on the point at which the interoperability support is handled.

For the first classification, interface differences, we identify two general categories:

- The *interface bridging* approaches bridge the differences between interfaces. They are characterized by the notions of offered and requested interface and define an interface transformation language. The interface transformation language requires the existence of two interfaces and allows one to express how the offered (requested) interface can be transformed to the requested (offered) interface. Note that the interface transformation language is programming language dependent.
- The *interface standardization* approaches standardize the interface under which a service (functionality) is offered. They are characterized by an interface definition language that allows one to express in a programming language independent way a specific interface. From the abstract definition of an interface a compiler will produce the necessary stub-interface in the implementation language selected. The compiler will always generate the same stub-interface for the selected target programming language.

For the second classification depending on the point at which interoperability support is handled, we also identify two categories:

- The *procedure-oriented interoperability* approaches that handle interoperability at the point of the procedure call.
- The *object-oriented interoperability* approaches that handle interoperability at the point of the object.

In the rest of this chapter we present a brief overview of some representative projects from different interoperability approaches, discussing their advantages and disadvantages, and describe in detail the object-oriented interoperability approach of the Cell framework [12].

3.2 Procedure-Oriented Interoperability

The problem of interface matching between offered and requested services has been identified by many researchers [6][15][18][21][22][25][26] as an essential factor for a high-level interoperability in open systems (see also chapter 12). Nevertheless, most of the approaches taken in the past 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 iteroperability (POI)*. In POI 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* [21] of the actual parameters passed to the procedure call at the client side to the requested procedures at the server side.

3.2.1 Interface bridging

An example of this approach is the one taken in the *Polylith* system [21]. 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 of parameters, e.g. five integers to an array of integers, parameter evaluation, e.g.the transformation of the strings "male" and "female" to integer values, and parameter extensions, i.e. 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.

3.2.2 Interface standardization

Whereas NIMBLE focuses on bridging the differences between the offered and requested service interfaces, the *Specification Level Interoperability (SLI)* support of the *Arcadia* project [25] 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*"[25]. 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. With SLI the implementer of a new service will need to specify the service interface with UTM 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 or her 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.

An approach similar to SLI has been taken in the *Common Object Request Broker Architecture* (CORBA) [18] of the Object Management Group (OMG). The Object Request Broker (ORB) "*provides interoperability between applications on different machines in distributed environments*"[18] 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, the IDL compiler will generate interface stubs for the client language through which the service can be accessed using the predefined language bindings.

3.2.3 Advantages and Disadvantages

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. 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 interrelated 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 interrelation 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 (i.e. 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 an 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 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 type*s. Composite non-migratable abstract types, like a database or keyboard type, cannot be passed as procedure parameters. This nevertheless 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.

The need for allowing non-migratable objects as parameters to operation calls was identified in the CORBA and a special data type was introduced called *object reference*. CORBA object references are data types that encapsulate a handle to a (non-migratable) object and are globally valid. However object references are a low level primitives which must be explicitly referenced and de-referenced by the server and the client. A higher-level primitive allowing direct access to object is clearly needed if we wish to have consistent access in an object-oriented environment.

3.3 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 into 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 object where the rectangle can represent a window into which the characters that represent the user/machine interactions are printed. In object-oriented environments it is the overall functionality of the object that is of importance and not the functionality of the independent operations. We call this type of interoperability where the semantics of the objects as a whole are preserved *object-oriented interoperability* (*OOI*).

3.3.1 Interface Bridging

An example of interface bridging in object-oriented interoperability is the one provided by the Cell framework [12] (where the concept of OOI was also introduced). The Cell is a framework for the design and implementation of "strongly distributed object-based systems". The purpose of the Cell is to allow objects of different independent object-based systems to communicate and access each other's functionality regardless of possible interface differences. That is, the same functionality can be offered with a different interface from different objects found either on the same or on different environments. The bridging of the interface differences is done via the *Interface Adaption Language (IAL)*. From the specification given in the IAL a compiler generates the required stub objects that support the requested interface and translate the incoming operation invocations to the invocations of the target object interface.

A more detailed presentation of the Cell interoperability approach is given in section 3.5.

3.3.2 Interface Standardization

The most important example of interface standardization in object-oriented interoperability is version 2 of CORBA. In contrast to the first version of CORBA, which was oriented towards C and C procedure calls, the second version is oriented towards a C++ environment and objects. Otherwise the functionality of CORBA and the basic elements are the same as described in section 3.2.2.

3.3.3 Summary

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 appli-

cations 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 IB-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 table 3.1 we give a summary of the different approaches presented above and their position in the two classifications.

	Procedure-oriented interoperability (POI)	Object-oriented interoperability (OOI)
Interface standardization (IS)	SLI, CORBA v. 1	CORBA v. 2
Interface bridging (IB)	NIMBLE	Cell

Table 3.1 *Classification of interoperability support approaches.*

3.4 Comparison of Interoperability Support Approaches

The interface bridging approaches provide a more general solution than the interface standardization approaches for the access and reuse of objects from different programming environments since they do not enforce any specific interface. The application designer can choose 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 the interface bridging approaches is that they make 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 they provide flexibility in the interconnection of diverse environments based on different models and abstractions.

One of the disadvantage of the interface bridging approaches comes from the fact that they do not enforce a common global representation model for expressing the interoperability bindings. Each execution environment is free to choose its own language. As a result the interoperability interface adaption specifications for a server need to be defined independently by the programmer for each execution environment in an interface adaption language that is specially tailored for the programming languages of the two environments. 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 [16] and KAROS [4], their IDL-based interoperation will not be able to use transactions.

Object-oriented interoperability and procedure-oriented interoperability approaches cannot be directly compared since they are designed for different programming environments: the first for object-oriented environments and the second for non-object-oriented environments. Nevertheless OOI is a generalization of POI using at its lower levels the same mechanisms as POI. Thus the major advantage of OOI over POI is that it can be applied as well to both types of programming environments and serve as bridge between object-oriented and non-object-oriented environments.

Although the interface bridging and interface standardization approaches are distinct in the way they approach the interoperability problem, they are not exclusive. An interoperability support system can very well support both approaches and give the programmers maximum flexibility in the reuse and access of objects in different programming environments. As an example we can consider CORBA which is an interface standardization interoperability support system. In a large CORBA-based open distributed system it will be difficult for all service providers to agree on a common interface for the servers they develop. As a result a number of different server interfaces will be available providing the same or similar services. However, applications being developed to access a specific server interface will not be able to access any other server even if the interface differences are minor. In addition, since it is not possible to anticipate the interfaces of future servers, applications will not be able to take advantage of newer, more advanced services. What is needed is to introduce interface bridging interoperability support. This can be easily done with the introduction of an *interface adaption* service that will allow a client to adapt its requested service interface to a specific offered interface and dispatch the service requests accordingly.

3.5 Interface Bridging — Object-Oriented Interoperability

We identify two basic components necessary for the support and implementation of interface bridging OOI (IB-OOI): *interface adaption* and *object mapping*. Interface adaption provides the means for defining the relations between types on different execution environments based on their functionality abstraction, and object mapping provides the runtime support for the implementation of the interoperability links.

3.5.1 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 ac-

tual interface of the server. In addition we will use the term *node* to specify the execution environment of an application (client or server), e.g. the Hybrid [7] execution environment or the Smalltalk [5] execution environment. In this sense a node can span over more than one computer, and more than one node can coexist 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. By 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 physical person who interacts and maintains the interoperability support system.

3.5.2 Interface Adaption

In a strongly distributed environment [24] 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 accessed services, 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. IB-OOI 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 interface adaption problem consists of defining and realizing the bindings and transformations from the interface that the client uses (requested interface), to the actual interface of the service (offered interface).

Ideally we would like to obtain an automatic solution to the interface adaption 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 interface adaption problem will require, at some point, human intervention since the system can automatically deduce neither which type matches which, nor 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 interface adaption 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.

3.5.2.1 Type Relations

In IB-OOI we distinguish three kinds 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, i.e.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 IB-OOI since we wish to support interoperability through the reuse of existing types. Thus in IB-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 a common 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 IB-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.

Although data types are the best candidates for an equivalence 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 is sufficient to allow the migration of their instances. In migrating an object to its equivalent on the remote node, the IB-OOI support must handle the representation differences of the transferred data. In

this sense the *type equivalence* of IB-OOI corresponds to *representation level interoperability* [25].

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 [17] 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 IB-OOI this type relation is defined as *type translation*.

Translation of the local type to the remote type is done with a user-definable translation function. This way the particularities of the semantic equivalence can be handled in a 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 — the first when the ARRAY OF CHAR represents a character string and is translated to a string, and the second when the ARRAY OF CHAR represents a collection of characters that need to be treated independently and which is translated to a Hybrid array of integer (in Hybrid characters are represented via integers).

Type translation can be compared to specification level interoperability, 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 interface adaption problem (i.e. 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, when for example one part of the requested functionality is offered from one type and another part from a second type, and uni-type matched types are the ones that are bound to a single type on the remote node.

The target of IB-OOI is to allow access to objects on remote nodes. The basic assumption being that the object in question cannot be migrated to 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 to the remote node or not. Parameter objects that cannot be migrated to 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

---

* CooL is a an object-oriented language designed and implemented in the ITHACA ESPRIT [20] project

as equivalent in the C++ node an int (integer) (Booleans in C++ are represented by integers), while the reverse is, in general, false.

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

Type matching is a general mechanism for interoperability support and can be used in all cases in place of equivalence and translation of types. However, the existence of translation and equivalence 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 always has the possibility of accessing any object locally, even an integer if this is needed, as might be the case with an integer that is stored at a specific memory address which is hard-wired to an external sensor (like a thermometer) and which is 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 IB-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 or 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 IB-OOI approach is that in the above scenario the application prototype will not be modified when local implementations of types are introduced* and the type relations change. The new type relations are introduced in the IB-OOI support and do not affect the application programs.

3.5.3 Object Mapping

Whereas interface adaption 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.

---

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

3.5.3.1 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 IB-OOI we call an *inter-object*. An inter-object differs from a *proxy*, as this is defined in [23], in three important respects. 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, i.e. 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 simply provides 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 (i.e. the implementation of a type) of the inter-object is created by the object mapper from the type match information and we call it an *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.

3.5.3.2 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 invoking the operations of the inter-object, passing other objects as parameters. IB-OOI allows objects of any type to be used as parameters at operation calls. The object mapper will handle the parameter objects according to their type relations with the remote node. This way objects for which an equivalent or translated type exists 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 implementation 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.

3.6 Interface Adaption

Expressing the relations and transformations between two (or more) interfaces can be done using a language which we call *Interface Adaption Language (IAL)*. IAL, just like

the existing interface definition languages (like the CORBA IDL) that allow the expression of an interface in an abstract language independent way, allows the expression of the relations and transformations required for the adaption of one interface to another in an abstract language independent way.

An IAL for the object-oriented interoperability support of the Cell framework prototype [8][9][11] was designed and implemented at the University of Geneva. The main goal of the Cell framework is to allow the objects of a node transparently to access and use services found on other heterogeneous nodes using the OOI support. IAL allows the user to express the interface relations between object types of the different nodes. The syntax of the IAL is very similar to the Hybrid language syntax [7][10][17], in which the Cell prototype was implemented.

In the rest of this section we give an overview of the implemented IAL using examples for the adaption of interfaces between Hybrid object types and CooL [1] object types. A complete description of IAL can be found in [13].

3.6.1 Type Relations

A type relation in IAL is defined for a specific remote cell which is identified by its name. For the examples given below 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.

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

In the definition of translated types we specify a translation function, like string2arrayOfChar and arrayOfChar2string, which performs the data translation.

```
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 3.1 *Hybrid type windowServer.*

3.6.1.2 Type-Matched Types.

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 the target type will be instantiated 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. In IAL this is noted with the addition of @ at the of remote type name:

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

3.6.2 Description of the Running Example

In order to describe the IAL syntax we use as examples a Hybrid type windowServer and a CooL type WINDOW_CONTROL. The Hybrid windowServer defines in the Hybrid cell the interface through which a window server is to be accessed (requested interface), while the CooL WINDOW_CONTROL provides an implementation of a window server (offered interface). For simplicity we assume that the operation names of the two types describe accurately the functionality of the operations. That is, the operation named newWindow creates a new window, while the operation get_Position returns the position pointed to by the pointing devices.

The Hybrid type windowServer (figure 3.1) has five operations. Operations newWindow and newSquareWin return the id of the newly created window or zero in case of failure. Op-

* The syntax of the operation bindings is described in detail in section 3.6.3.

```
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 3.2 *CooL type WINDOW_CONTROL.*

eration refreshDisplay returns true or false, signifying success or failure. Operation readCoordinates returns the coordinates of the active point on the screen as read 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 3.2) has four methods. The methods create_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) as adapted by the scaling factor.

3.6.3 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 the IAL we anticipate two levels of interface differences — first in the required parameters (order, type, etc.) and second in the set of supported operations, i.e. different number of operations with aggregated, segregated or slightly* different functionality. The resolution of these differences corresponds to the parameter and interface phases of the interface adaption definition.

3.6.3.1 Parameter 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.

* The term is used loosely and it is up to the user to define what is a “slight” difference in functionality.

- *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.

The IAL 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 IAL as follows:

```
newWindow : create_win($3, $4, $1, $2, 17 ) ^ 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 is an integer that specifies the colour of the new window. This information does not exists in the offered parameters. Nevertheless, in this case, we can use a default value using a integer literal, like in the example the number 17. The returned value from create_win, noted as RET in IAL, 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 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, 17) ^ RET;
```

where functions bottomRX and bottomRY are adaption functions. Adaption functions are user-defined functions, private to the specific interface adaption. 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 the IAL the adaption functions are included at the end of the interface adaption 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 (topLeftY - side ) ; }
@}
```

The adaption functions will be invoked locally (i.e. 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, i.e. 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 follows:

```
@{
    int2bool : ( integer : intval ) -> boolean ;
        { return ( intval ~=? 0); }
@}
```

Multi-type mapped parameters

In IAL 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 to 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 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 ;
```

3.6.3.2 Interface Phase

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 operation. Adapting a requested operation 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, IAL 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.

3.7 Object Mapping

Whereas interface adaption provides the means to express in an implementation language-independent way the relations between heterogeneous interfaces, object mapping provides the required language-dependent run-time interoperability support. The first task

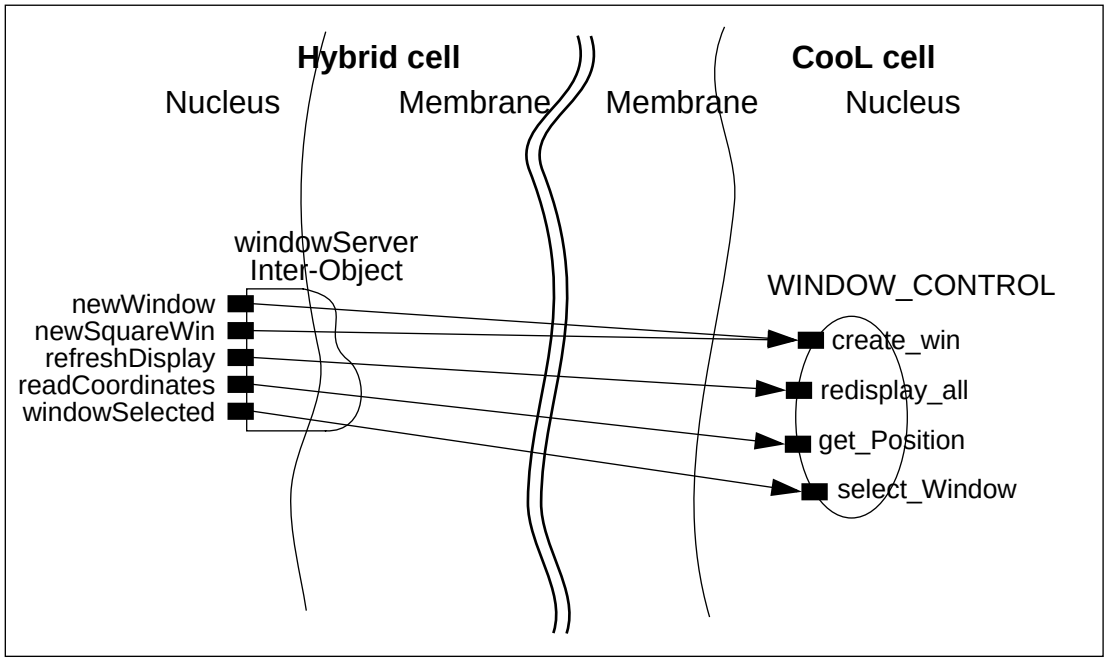

Figure 3.3 *Object mapping.*

of object mapping is to generate from the interface adaption specifications the inter-classes at the client side. Instances of an inter-class provide the client with the requested service interface and their principal task is to forward the operation invocation to the target server according to the specified interface transformations and adaptions.

In the following we describe the functionality of object mapping via the previously described example of interface adaption between the Hybrid WindowServer and the CooL WINDOW_CONTROL. In figure 3.3 we present the binding between the operations of the Hybrid inter-object and the CooL server and 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 3.4) — and which is bound to the method get_Position.

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

From the four parameters passed to operation readCoordinates, the first three (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. The object mapping server will thus instantiate on the CooL cell two objects: 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 3.5).

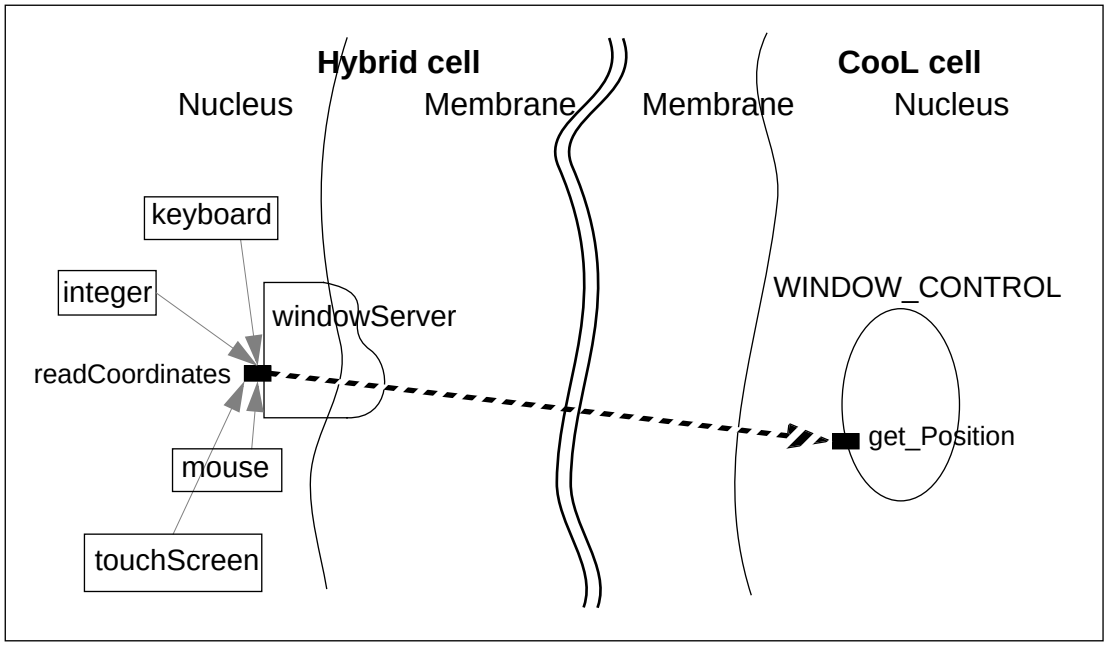

Figure 3.4 *Operation call forwarding.*

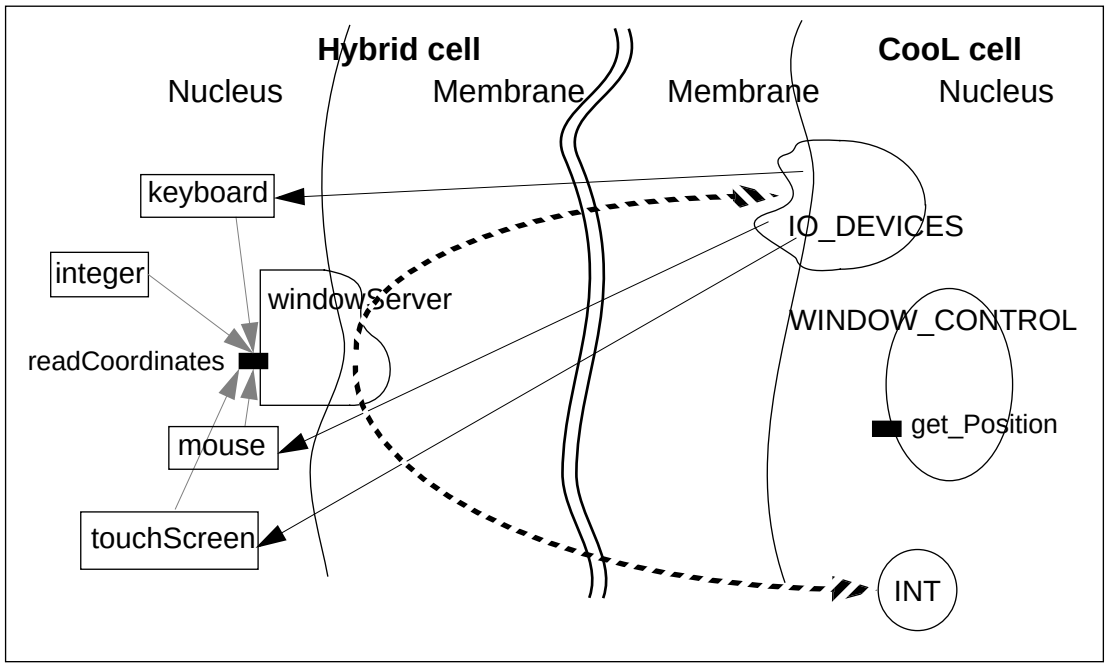

Figure 3.5 *Parameter transfer.*

When the transfer of the parameters has been completed the object mapping server will proceed with the invocation of the remote operation. The operation get_Position will be invoked with the IO_DEVICES inter-object and the INT object (figure 3.6) as parameters. The

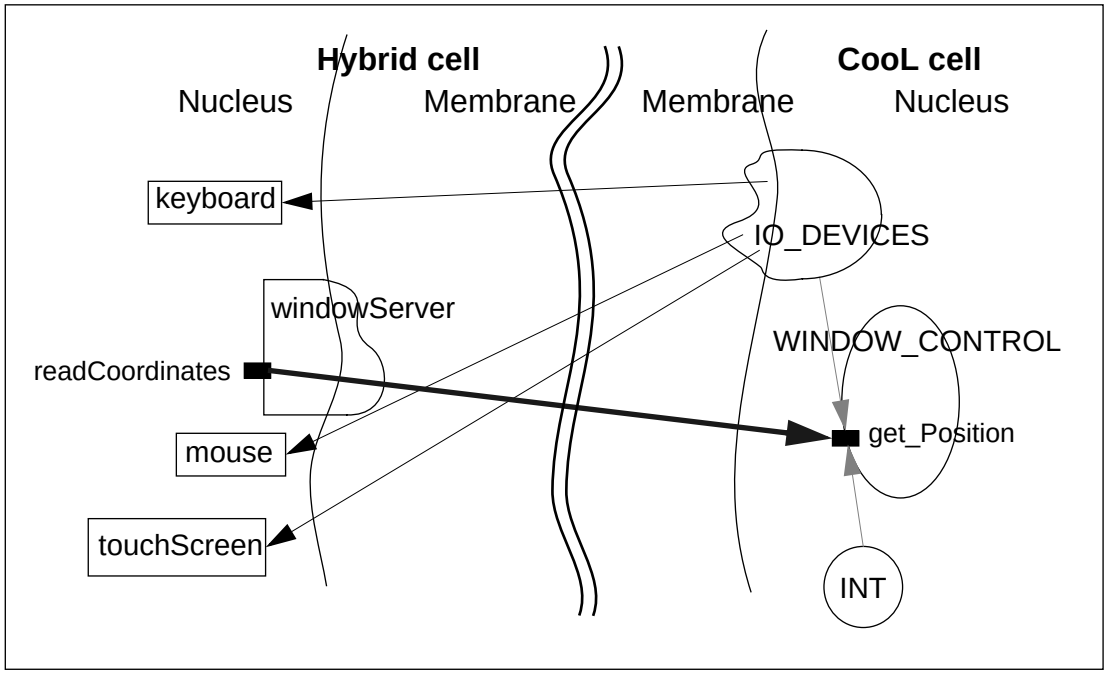

Figure 3.6 *Remote operation invocation.*

result, an object of type POSITION, will then need to be returned to the Hybrid caller. Because for the CooL type POSITION there exists a translation to the Hybrid type point, the object mapping server will instantiate an object of type point on the Hybrid cell which will be initialized to the translated value of the POSITION object. This object will be the result returned to the caller of the readCoordinates operation.

During the transfer of parameters the object mapping server 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. In this case when the instantiation of an IO_DEVICES inter-object is requested, the type-matching server will dynamically request the definition of the type match. The user will be required to define on the fly a type match for the IO_DEVICES type. Once this is done the object-mapping server will resume the transfer of the parameters. This way an application can be started even without any type relations defined. The object-mapping server will prompt the user to define all needed type relations during the first run of the application.

3.8 Conclusions and Research Directions

One of the important advantages of object-oriented design and development methodology is the ability to reuse existing software modules. However, the introduction of many programming languages with different syntaxes, semantics and paradigms severely restricts the reuse of objects programmed in different programming languages. Although *adhoc*

solutions can be given to solve specific inter-language reuse cases, different interoperability support methods provide the framework for consistent inter-language access and reuse of objects.

We classify the interoperability support approaches in two ways: first depending on the way that they solve the problem of the different interfaces, and second on the point at which the interoperability support is handled. For the first classification we distinguish the *interface standardization* approaches, which standardize the interface under which a service (functionality) is offered, and the *interface bridging* approaches, which bridge the differences between interfaces. For the second classification we distinguish the *procedure-oriented interoperability* approaches, which handle interoperability at the point of the procedure call, and the *object-oriented interoperability* approaches, which handle interoperability at the point of the object.

From the above approaches the interface bridging object-oriented interoperability (IB-OOI) approach is the most flexible one since it does not impose predefined interfaces and can be applied equally well to both object-oriented and non-object-oriented environments. The Cell framework, which we describe in detail, provides an example of the IB-OOI approach.

Because the IB-OOI is by no means incompatible with other interoperability approaches, its ideas and concepts can be incorporated into other interoperability frameworks, e.g. the CORBA, and significantly enhance their openness and interoperability support. Furthermore the flexibility and generality of the IB-OOI ideas can provide a framework for the solution of software integration and software evolution problems related to legacy systems.

3.8.1 Openness of Interoperability Platforms

One of the major disadvantages of existing interoperability frameworks, the most prominent of which is CORBA, is that they are *closed to themselves.* That is, client and server applications interacting via the interoperability platform must be implemented making use the specific platform interfaces. As a result, taking CORBA as an example, existing applications cannot be incorporated in the CORBA "world" (non-CORBA clients cannot use CORBA services, and non-CORBA servers cannot offer their services to CORBA clients), nor can CORBA applications be moved to a non-CORBA environment.

Designing an interface adaption service for CORBA that will allow C++, for example, client applications to access CORBA services via their IDL interface will significantly enhance the openness and acceptability of CORBA and will allow almost any application to take advantage of the services CORBA offers.

3.8.2 Interoperability and Legacy System Migration

One of the major problems that companies are facing due to the rapid advances of the computer software and hardware technologies is the migration of their legacy systems to a new

platform. Most of the given solutions are *adhoc* case-dependent solutions; only recently has some kind of methodology started appearing [2][3]. However, although the problem of legacy system migration is in effect an interoperability problem, it has not been recognized as such. The reason is that most of the work and research done in the area of interoperability support focuses on the interoperability support of new applications using the interface standardization approach and does not consider existing legacy applications.

A prominent framework for the support of legacy system migration can be provided with the interface bridging object-oriented interoperability (IB-OOI) approach. A smooth incremental migration of a legacy system can be achieved by identifying its components and their interfaces and using an IB-OOI support to replace the legacy components with new ones, which most probably have a different interface [14]. This way new components can be incrementally added to the system without affecting the remaining legacy ones.

References

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

[2] Thomas J. Brando and Myra Jean Prelle, "DOMIS Project Experience: Migrating Legacy Systems to CORBA Environments," Technical Report, The MITRE Corporation, Bedford, Mass., 1994.

[3] Michael L. Brodie and Michael Stonebraker, "DARWIN: On the Incremental Migration of Legacy Information Systems," DOM Technical Report, TR-0222-10-92-165, GTE Laboratories Inc., March 1993.

[4] Rachid Guerraoui, *Programmation Repartie par Objets: Etudes et Propositions*, Ph.D. thesis, Universite de Paris-Sud, Oct. 1992.

[5] Adele Goldberg, *Smalltalk-80*, Addison-Wesley, Reading, Mass. 1984.

[6] Yoshinori Kishimoto, Nobuto Kotaka, Shinichi Honiden, "OMEGA: A Dynamic Method Adaption Model for Object Migration," Laboratory for New Software Architectures, IPA Japan, Working Paper, April 1991.

[7] Dimitri Konstantas, Oscar Nierstrasz and Michael Papathomas, "An Implementation of Hybrid," in *Active Object Environments*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1988, pp. 61–105.

[8] Dimitri Konstantas, "Cell: A Model for Strongly Distributed Object Based Systems," in *Object Composition* ed. D. Tsichritzis, CUI, University of Geneva, 1991, pp. 225–237.

[9] 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, Oct. 17–18, 1991, pp. 156–163.

[10] Dimitri Konstantas, "Hybrid Update," in *Object Frameworks*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1992, pp. 109–118.

[11] Dimitri Konstantas, "Hybrid Cell: An Implementation of an Object Based Strongly Distributed System," *Proceedings of the International Symposium on Autonomous Decentralized Systems ISADS '93*, Kawasaki, Japan, March 1993.

[12] Dimitri Konstantas, "Cell: A Framework for a Strongly Distributed Object Based System," Ph.D. thesis No. 2598, University of Geneva, May 1993.

[13] Dimitri Konstantas, ''Object-Oriented Interoperability,'' *Proceedings ECOOP '93*, ed. O. Nierstrasz, *Lecture Notes in Computer Science*, vol. 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 80–102.

[14] Dimitri Konstantas, "Towards the Design and Implementation of a Safe and Secure Interoperability Support Layer in CHASSIS," CHASSIS SPP project technical report, University of Geneva, March 1994.

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

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

[17] Oscar Nierstrasz, ''A Tour of Hybrid — A Language for Programming with Active Objects,'' *Advances in Object-Oriented Software Engineering*, ed. D. Mandrioli, B. Meyer, Prentice Hall, 1992, pp. 167–182.

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

[19] Object Management Group, O*bject Management Architecture Guide*, OMG TC Document 92.11.1, Revision 2.0, Sept. 1992.

[20] Anna-Kristin Pröfrock, Dennis Tsichritzis, Gerhard Müller and Martin Ader, "ITHACA: An Integrated Toolkit for Highly Advanced Computer Applications," in *Object Oriented Development*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, July 1989, pp. 321–344.

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

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

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

[24] Peter Wegner, ''Concepts and Paradigms of Object-Oriented Programming,'' *ACM OOPS Messenger*, vol. 1, no. 1, Aug. 1990, pp. 7–87.

[25] 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.

[26] Daniel M. Yellin and Robert E. Strom, "Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors," proceedings of the 9th annual conference on *Object Oriented Programming Systems, Languages and Applications — OOPSLA'94*, Portland, Oreg., 23–27 Oct. 1994.

Annex I: Interface Adaption 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
| INTEGER_LITERAL

localType : STRING

remoteType : STRING

remoteOpName : STRING

remoteCellId : STRING

transFunction : STRING

adaptFunct : STRING

Program : *Program code in Native Language.*

Annex II: Type Match Definition Example

```
CooLCell :: windowServer -> WINDOW_CONTROL  {
    newWindow : create_win($3, $4, $1, $2, 17 ) ^ RET ;
    newSquareWin : create_win ( bottomRX($1, $3), bottomRY($2, $3), $1, $2, 17 )
            ^ 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, 1)
         )  ^ RET ;
}
@{
    bottomRX : (integer : topLeftX, side ) -> integer ;
        { return (topLeftX + side ) ; }

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

    int2bool : ( integer : intval ) -> boolean ;
        {
            return (intval ~=? 0) ;
        }
@} ;
```