

# Portable Serialization of CORBA Objects: a Reflective Approach

Marc-Olivier Killijian  
LAAS-CNRS

7, Av. du Colonel Roche  
31077 Toulouse Cedex 4 (France)  
+33.5.61.33.62.41  
marco.killijian@laas.fr

Juan-Carlos Ruiz  
LAAS-CNRS

7, Av. du Colonel Roche  
31077 Toulouse Cedex 4 (France)  
+33.5.61.33.69.09  
juan-carlos.ruiz-garcia@laas.fr

Jean-Charles Fabre  
LAAS-CNRS

7, Av. du Colonel Roche  
31077 Toulouse Cedex 4 (France)  
+33.5.61.33.62.36  
jean-charles.fabre@laas.fr

## ABSTRACT

The objective of this work is to define, implement and illustrate a portable serialization technique for CORBA objects. We propose an approach based on reflection: through open compilers facilities the internal state of CORBA objects is obtained and transformed into a language independent format using CORBA mechanisms. This state can be restored and used by objects developed using different languages and running on different software platforms. A tool was developed and applied to a *Chat* application as a case study. The proposed technique is used to exchange state information between a C++ and a Java incarnation of this CORBA service. An observer tool enables the object state to be displayed and analyzed by the user. The applicability of this technique to various domains is discussed. Beyond the interest of language reflection, we finally advocate that operating system and middleware reflection would also be powerful concepts to extend the work presented in this paper.

## Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming;  
E.2 [Data storage representations]: Object representation; H.3.4  
[Systems and Software]: Distributed Systems

## General Terms

Algorithms, Design, Languages, Reliability, Portability.

## Keywords

CORBA, Serialization, Reflection, Open compilers

## 1. INTRODUCTION

Handling the state of CORBA objects is a crucial issue in many application domains: migration, persistence, object replication, etc. For interoperability reasons, obtaining and restoring the state of individual CORBA objects should be done in a language independent way so that the state can be exchanged between different incarnations of the same object,

resulting from different programming languages. We use in this paper the terms *portable* and *language (and platform) independent* interchangeably.

Serialization in a homogeneous environment has been studied quite extensively, e.g. in [1] [2]. This is not the case of serialization in heterogeneous environments where works like [3] [4] focus on platform interoperability but not on language portability. Our objective in this work is to provide facilities to serialize and de-serialize CORBA objects in a portable format. We introduce an abstract model of object state so that a serialized state is interoperable and portable.

The state of a CORBA object is quite complex and comprises several facets, such as the attribute facet, the platform facet and the communication facet. The attribute facet includes mostly all internal object state variables structured using object-oriented programming features (data types, inheritance, composition, etc.). The platform facet includes all internal data of the underlying software platform (middleware and operating system layers), some of which is application object dependent. The communication facet relates to all the messages in transit, the state of the protocol stack, etc.

This paper concentrates on the serialization of the attribute facet of a CORBA object. The other facets have been handled in the past using conventional facilities (e.g. trap of system calls, message logging, end-to-end protocols, etc.) or require other reflective facilities that are out of the scope of this paper. We will back to these facets in Section 5.3.

The approach that we propose for handling CORBA object serialization is CORBA-compliant, transparent to application programmers and portable. These requirements are met by using both open compilers and the generic runtime support supplied by CORBA. Open compilers apply the notion of reflection [5] at compile-time in order to provide facilities for customizing the compilation process of a program. These facilities are exploited for automating the analysis of object definitions and generating (according to the results of that analysis) adequate mechanisms for serializing and de-serializing CORBA object states. This solution has three major benefits. First, it minimizes the effort required for providing customized implementations of serialization; the analysis and generation rules supplied to the open compiler are defined only once and they can be later used on any CORBA object. Second, these rules are automatically applied; this avoids the participation of unskilled programmers in the generation of the serialization mechanisms. Third, the technique only depends on abstractions supplied by the CORBA support that are, by definition, both platform and programming language independent. This makes possible the provision of a language

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA '02, November 4-8, 2002, Seattle, Washington, USA.

Copyright 2002 ACM 1-58113-471-1/02/0011...\$5.00.

independent representation of objects state and then the provision of portable serialization for CORBA objects. Although this approach could have been implemented using dedicated compilers, we illustrate in this paper the benefits of using open-compilers instead.

The next section recalls the basic notions of reflection and open compilers used all through the rest of the paper. Section 3 first provides a high-level view of the proposed approach. Then, it addresses the definition of a CORBA object state model that is later applied on CORBA objects using open compiler facilities. The resulting technique is then exemplified, and its benefits illustrated, in the Section 4 using a C++ and a Java implementation of a CORBA *Chat* service. Section 5 discusses the applicability of the approach to several domains like mobile agents and fault-tolerance. Section 6 comments related work and discuss the pros and cons of the solution. Finally, Section 7 presents the conclusions.

## 2. Reflection and open compilers

Computational reflection is the activity performed by a system when doing computation about its own computation [5]. This notion enables a system to be structured in two layers: the *base-level*, executing the application components, and the *meta-level*, running components devoted to the implementation of requirements that are orthogonal to the application. In most object-oriented reflective systems, a so-called MetaObject Protocol (MOP) handles interactions between the base- and the meta-level.

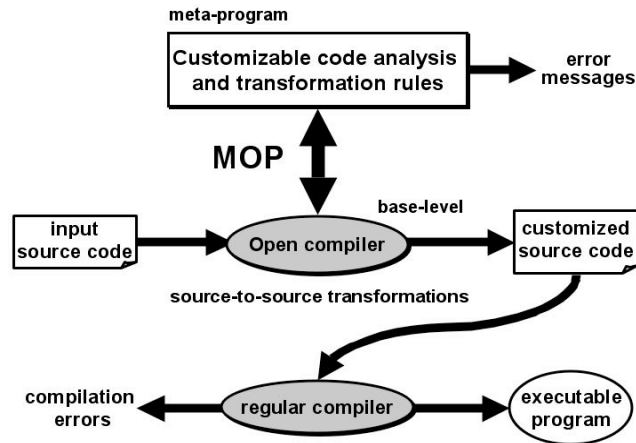


Figure 1. Open compilation process

Open compilers, like [6] [7], apply the notion of reflection at compile-time in order to open the compilation process of a program. Essentially, these compilers are macro-systems providing means to perform source-to-source transformations. Figure 1 provides a high-level view of the compilation process proposed by this type of compilers. The base-level of an open compiler encapsulates the work typically performed by a conventional compiler. The observation and control facilities supplied by the open compiler provide the required means to observe the program structure, reason about it and (eventually) act on its translation. These facilities correspond to a compile-time MOP. The *meta-program* uses these MOP facilities for defining rules that (1) analyze the structure of the input program, and (2) transform this structure according to the needs. From now, these rules will be referred as *analysis and*

*transformation rules*. It is worth noting that meta-programs may also generate error messages. When no error message is generated by the meta-program, the customized code finally produced can be compiled using a regular compiler.

As Gregor Kiczales states in [8], "*aspect-oriented programming (AOP) has a deep connection with work in computational reflection and metaobject protocols*". Conceptually, AOP promotes separation of concerns in modern programming languages. In practice, reflection is a powerful mean to reach that goal as shown in [9]. Using the AOP terminology, an open compiler can be defined as a weaver tangling the aspect code of a meta-program with the non-aspect code supplied by a basic program. The join-point model used in AOP to specify when the aspect code can be activated is in that context expressed in terms of a MOP. Through this MOP, meta-programs analyze and customise input programs according to the needs of the considered aspect. This is how our approach exploits the open compiler technology, defining meta-programs implementing a "*portable serialization*" aspect for CORBA objects. Since these objects can be implemented in many different languages, the analysis and transformation rules specifying the aspect must be also specialized for each considered target language. The following Sections focus on how the "portable serialization" aspect can be defined in general, and mapped to different programming languages.

## 3. APPROACH

### 3.1 Overview

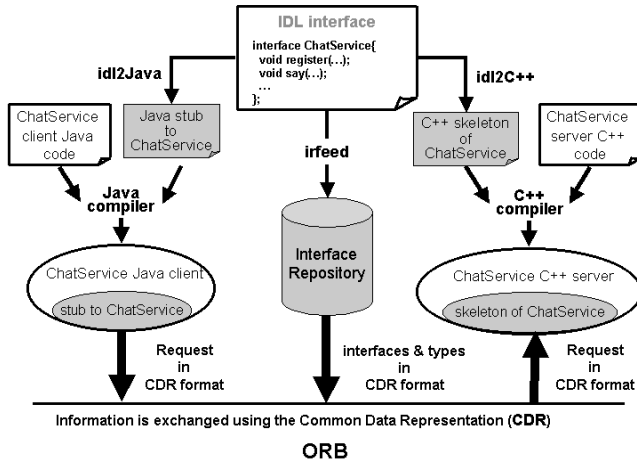
*CORBA* [10], the acronym for Common Object Request Broker Architecture, is the Object Management Group (OMG) solution for distributed object computing. The major benefit of using CORBA relies on the support that it provides for the development of third-party applications, which are able to interoperate despite the programming language used for their implementation, the operating system on which they run and the underlying platform they use to communicate.

The object model proposed by CORBA is based on a client-server paradigm where servers' interfaces are defined using the standard *Interface Definition Language* (IDL). This language is independent of programming languages, but is mapped to all the popular languages (Java, C++, etc.). A so-called IDL compiler automates that mapping. Typically, IDL compilers generate two types of components: (i) *stubs*, which are server proxies used by clients; and (ii) *skeletons*, which are used by servers for handling incoming client invocations. Messages exchanged between clients and servers are formatted using a platform independent octet stream representation, which is called *Common Data Representation* (CDR). In order to maintain the relationship between each CDR data and its original IDL data type, CORBA specifies a set of transforming rules for the formatting of IDL types in CDR. These rules solve interoperability problems of variable byte addressing and data alignment among heterogeneous platforms. The marshalling process is the one enabling mapping from each platform or language format to the CDR format. The symmetric process is called un-marshalling.

The separation promoted by the IDL and the CDR format between interfaces, implementations and platforms, is the essence of CORBA, i.e., how it enables interoperability. Servers export to the CORBA world their interfaces through the *Interface Repository* (IR) service. This is the component of the

architecture that provides runtime support for the IDL type system. This runtime type support is essential to ORBs in order to check the integrity of the CDR-formatted messages exchanged between clients and servers. On the other hand, CORBA objects (clients or servers) have also access to the IR. It is worth noting that this type support is generic and thus language and platform independent.

Figure 2 shows how C++ and Java interoperate provided the IDL definition of a *ChatService* interface. First of all, the IDL is stored in the IR using the *irfeed* tool that each ORB supplies. Then, any CORBA object can discover what is a *ChatService*. The clients use *stubs* in order to communicate with the servers. The servers' *skeletons* interpret clients' requests and activate accordingly the adequate server's method. The IDL compilers (*idl2Java* and *idl2C++* in our example) produce the stubs and the skeletons.



**Figure 2. High-level view of a CORBA service (example of the Chat service)**

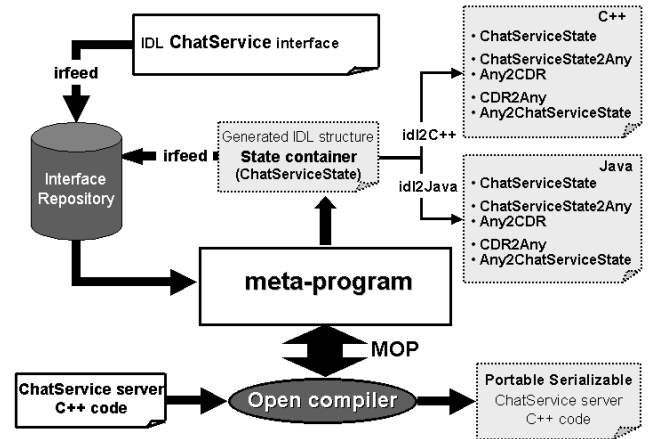
The approach that we define in this section proposes to serialize the *attribute facet* of CORBA objects using a CDR representation. It introduces the notion of *CORBA object state container* that enables object serialization to benefit from the generic infrastructure defined in CORBA. From a design viewpoint, this concept can be defined as an IDL container able to “hold” the complete state of a CORBA object. For the time being, CORBA does not provide any tool well suited for automating the generation of CORBA state containers from object implementations (as discussed in section 6). Hence, we propose the use of open compiler facilities to that goal.

Our approach uses open compilers for analyzing CORBA object implementations and generate accordingly (i) the adequate state containers and (ii) the language-dependent mechanisms required for saving and restoring object states to/from these containers. First, this analysis and generation process is transparent to the application programmers. Second, the generated state containers can be mapped using IDL compilers to most common programming languages. This is how our solution exploits the existing CORBA support for providing portability.

At the CORBA level, the benefits of this approach are three-fold: (i) object state containers are automatically generated from implementations; (ii) through these containers, the notion of object state becomes typed, which promotes a more rigorous and type-safe management of object states at the

implementation level; (iii) every CORBA application is able to dynamically discover and handle the structure and contents of these containers using the general runtime type support supplied by the IR.

At the implementation level, the declaration of state containers in IDL has a major benefit: it enables the use of the CORBA *any* variables, *anys* in short, for handling object states. *Anys* are generic IDL data type value without loss of type information. Basically, an *any* encapsulates an un-typed data buffer where the IDL data value is stored with its associated type-code, which can be used to interpret the contents of the data buffer. In practice, IDL compilers are responsible for generating the mechanisms for packing and unpacking *anys* to/from IDL variables. In the same way, CORBA also formalizes the process of marshalling and un-marshalling *anys* to/from CDR-formatted buffers. All these mapping mechanisms are basic in our approach for the provision of portability. Through them, state containers can be manipulated using any CORBA-compliant programming language or platform. In other words, multiple implementations (possibly in different languages) of the same object can exchange their state, making it possible to migrate an object to an incarnation written in another language.



**Figure 3. Overview of the approach**

Figure 3 exemplifies the approach on the *ChatService* example introduced above. The meta-program has two main inputs: the code associated to the *ChatService* implementation and its interface definition, which is retrieved from the IR. Accordingly, the meta-program drives the open compiler in order to generate the adequate *ChatServiceState* container, which is stored in the IR. This state container is also compiled and mapped to both C++ and Java. This produces the necessary mechanisms for (i) packing/unpacking state containers to/from *any* variables (*ChatServiceState2Any* and *Any2ChatServiceState*) and (ii) marshalling/un-marshalling the resulting *anys* to/from CDR buffers (*CDR2Any* and *Any2CDR*). Finally, the meta-program appends to the implementation the necessary mechanisms for serializing/un-serializing *ChatService* instances to/from the portable *ChatServiceState* containers. However, it is worth noting that the user can, before the final compilation step, customize the “*Portable Serializable*” code generated in order to introduce some optimization or any complementary treatment.

## 3.2 A Structural Model for CORBA Objects State

In order to provide portable object serialization, we need a common object state model, valid for any programming language, so that the representation of the serialized objects can be interpreted within each programming language structural model. This section fixes the model that we consider for CORBA objects state. This model is basically defined in terms of (i) its type system and (ii) the object-oriented features that it is able to handle.

An object state, and in particular its attribute facet, can be defined as a set of internal variables (the object attributes) each one with a name, a value and a type. Our approach externalizes this compound of information to CORBA through state containers defined as IDL structures. These structures contain one field for each object attribute. The name of the field is the one of the object attribute. The type of the field is determined according to a mapping that must be defined between each programming language and IDL data types. The type of an attribute (and its mapping to IDL) defines the technique to be used in order to serialize the value of that attribute. An integer, for instance, cannot be handled like a string. The type system considered in our solution is the one defined by IDL. Data types included in our type system can be divided into basic types and constructed types.

**Basic types** are:

- *Built-in types* – The IDL built-in types are short, unsigned short, long, unsigned long, float, double, char, boolean and octet.
- *String types* – IDL defines a specific type for *Strings*
- *CORBA object reference types* – CORBA object references are identifiers used by clients to access distant CORBA objects.
- *Class types* – From a programming language viewpoint, class types are the basic notions for the provision of class *inheritance* and object *composition*. Handling inheritance and composition is, in practice, one of the major concerns tackled in the next section. It is worth noting that we support multiple interface and simple implementation inheritance. We elaborate on this issue in section 3.3.2.

**Constructed types** are data types defined in terms of one or more of the above basic types. We distinguish: *Structures*, *Arrays*, *Sequences* and *user-defined* types. Sequences are CORBA unbounded array types whose length is dynamically defined according to the number of elements packed in the sequence. User-defined types can be viewed as aliases to other constructed or basic types.

According to the CORBA object model, application programmers may encapsulate several objects implementing one or several CORBA interfaces inside a single CORBA object. Each of these objects is called a *servant*. For the time being, we only consider the existence of one servant per CORBA object. The main motivation for this is that our serialization approach needs a single root object in the implementation. Consequently, the state of a CORBA object is in fact the state of its incarnation: its unique servant. It is worth noting that this assumption does not prevent the encapsulation in a CORBA object of other internal objects not incarnating a CORBA interface. As a result, the serialization of a CORBA object leads to the serialization of its servant, which

provokes the serialization of each one of the internal objects associated to the servant.

*Encapsulation* is another object-oriented feature of great importance in our model that is enforced by CORBA. The notion of CORBA object promotes a strong encapsulation of data variables. For instance, a CORBA attribute can be defined in the public interface of a CORBA object, but in practice, this attribute will be private to that object and can be only accessed through special methods called *accessors*. This is very useful as it promotes independency among the states of different CORBA objects. As a result, state consistency is a problem that must be locally handled for each CORBA object. In our solution, class attributes and global variables are simply considered as attributes at the top of the object state hierarchy.

## 3.3 The Meta-Program

Open compilers are used to apply our object state model to particular CORBA object implementations. Concretely, open compilers are driven by a meta-program reasoning about (and customizing) object-oriented programs in terms of: the defined classes, their members (methods and attributes), the existing associations between the defined classes (established through inheritance, composition or delegation) and the manipulations performed on these notions by application programmers (like the instantiation of a class, the invocation of a method, the access to an attribute and so on). The use of open compilers, instead of plain compilers, enables to concentrate on the specific issues for which the compiler is used instead of complex issues of compiler development. The metaobject protocols of the open-compilers are largely sufficient for the issues we have to deal with and furthermore are really easy to use.

In our case, the open compiler activates the meta-program each time that a new class definition is detected in the input program. Then, the meta-program has three responsibilities: first, enforcing the **conformance rules**, i.e. checking the conformity of the class definition with the object state model; second, applying the **CORBA-mapping rules**, i.e. to build CORBA state containers from the supplied class; and finally, the **language support generation**, i.e. extending the class definition with the methods needed for handling CORBA state containers at the implementation level.

### 3.3.1 Conformance rules

These rules implement both structural and type checking. Their goal is to determine whether or not the classes fit the object state model specified in the previous section. We rely on the introspection facilities supplied by the open compiler in order to analyze class implementations. Type information supplied by the IR is also valuable for associating language data types with their respective IDL types.

**Type-checking** - In order to implement these rules, the meta-program needs to determine the type of each attribute. Basic types and strings are directly supported by the open compiler. On the other hand, in order to differentiate class types from CORBA object references we need a more sophisticated approach. From a structural viewpoint, CORBA object references are class types that (i) inherit (directly or indirectly) from CORBA::Object; and (ii) supply the methods defined in the CORBA interface of the server. These structural requirements can be verified on each class by inspecting its name, its super-classes, its methods and so on. However, the code inspection facilities supplied by the open compiler have

a limited granularity. Hence, the type support provided by the IR is essential in order to complete this analysis. Essentially, we check for each object reference type whether or not its associated CORBA interface exists in the IR. If this is not the case, the associated class type is considered as being a language type. Constructed type attributes are analyzed in two steps. First, we differentiate structures, arrays, sequences and type aliases. Then the data types contained in the constructed type attribute are recursively analyzed.

**Unique IDL mapping**– These rules forbid every language type without (or with more than one) equivalent IDL type. Indeed, for portable serialization, we need to be able to interpret a state non-ambiguously while the IDL specification is sometimes ambiguous. For instance, the “IDL to C++ mapping” specification [11] states that `CORBA::char` and `CORBA::octet` are the C++ types respectively associated to the `char` and `octet` IDL types. Conversely, a C++ `char` type can represent both `char` and `octet` IDL types. Then, the use of the C++ `char` type is ambiguous and is forbidden. In that case, the alternative is the use of the (unambiguous) C++ alias types `CORBA::char` and `CORBA::octet`. For the handling of built-in types and strings, we use a dictionary that contains, for each programming language, the mapping of the allowed language types. Typically, strings are directly mapped to languages built-in types, like in Java the `java.lang.string`. However, some other languages, like C++, do not have direct support for such data type. In C++, strings are usually handled using pointers to characters. Although this is conventional, it is also ambiguous and thus must be avoided. Regarding this particular issue, one can consider two solutions: first, every pointer to character attribute is considered as being of type string; or second, we rely on more adapted types, like the wrapping C++ type `CORBA::String_var` specified by the IDL mapping standard. Both solutions are allowed in our approach although the second is the most suitable.

**Inheritance** – As stated above, we support multiple interface inheritance and simple implementation inheritance. Indeed, multiple implementation inheritance can lead to serious problems for determining the state of an object. For example, let a class A inherits from classes B and C. At the same time, both B and C inherit from a common class D. This “diamond” inheritance tree leads to the need of serializing the attributes defined in class D twice when serializing the state of an A object. This issue concerns only some languages, like C++. In other languages, like Java, multiple implementation inheritance is not allowed.

In summary, these rules enforce a common object state model whatever the programming language used is. Using this common model, the state of an object can be safely exchanged between entities written using different OO languages. Obviously, these conformance rules have to be specialized according to the programming language considered; this will be discussed in section 4. When a conformance rule is violated, then the analyzed class does not conform to the defined object state model. In this case, the compilation process is stopped and a message identifying the problem is generated. This provides a useful feedback to fix the problem by choosing a more adequate programming alternative.

### 3.3.2 CORBA-mapping rules

These rules guide the mapping of object states to CORBA state containers. Attribute data types are mapped to IDL according to the following rules:

**Built-in types and strings** – The correspondence between each data type and its IDL type equivalent is determined through the data type dictionaries introduced in section 3.3.1.

**Object reference types** – Object reference types are mapped to IDL strings. According to the CORBA specification, every object reference can be stringified, i.e. saved in a string. On the other side, CORBA also standardizes the process of restoring object references from strings.

**Class types** – Classes are mapped to IDL structures. The name of the IDL structure generated for a class called A is *AState*. When a class A inherits from another one called B, then the IDL structure generated for class A (*AState*) contains a field of type *BState* in which the internal state of the B super-class is stored. Thus, inheritance is handled recursively. On the other hand, associations among class types are differently handled depending on the type of composition considered:

- Composition by value is handled by recursion. Remember that composition by value is when an object contains another object. Hence, the IDL structure of the container object will include one field for saving the state of the contained object.
- Composition by reference is when an object A contains the reference of an external object B. Then, A can use the service supplied by B, but A does not contain B, as in composition by value. Composition by reference is much more difficult to handle since it is a potential source of cycles in the object state graphs to be serialized. Consider, for instance, a circular list of objects. The serialization of such list requires the memorization of the object references already serialized in order to avoid (i) the serialization of the same object more than once, and (ii) an infinite recursion in the serialization process due to loops in the object graph being serialized.

We propose the IDL support defined in Figure 4, in order to handle the types of composition described above. The *ReferenceAndState* structure is used when an object is serialized for the first time. In this structure, we save both the object *reference* (identifier) and its *state*. Since it is not possible to anticipate the format of a state container, the *state* field of the considered structure is defined as an *any*. On the other side, the *Reference* structure is used for saving the *reference* of an object already serialized. In summary, the *ReferenceAndState* of an object will be saved in a *ClassType* only the first time that the object is serialized. Additional attempts for serializing the same object will result in saving only the *Reference* of the object.

```

struct ReferenceAndState{
    long ref;
    any state;
};

union ClassType{
    Reference ref;
    ReferenceAndState ref_and_st;
}

struct Reference{
    long ref;
};

```

Figure 4. IDL support for Class types

**Constructed types (structures, arrays and sequences)** – These types have a direct mapping to IDL. The basic types of each structure field and each sequence and array element are handled according to the CORBA-mapping rules defined in this section.

To conclude, one must understand the important role of the IR type support in handling the adequate mapping of object states to CORBA state containers. Queries to the IR are issued to determine whenever a type has already been defined in IDL. When it is not, the meta-program actually does the mapping. When it is, the IR furnishes the mapping.

### 3.3.3 Language support generation

The third role of the open compiler is to generate the mechanisms needed for handling CORBA state containers at the implementation level. First, the CORBA state containers generated by the CORBA-mapping rules are mapped to each considered language using standard IDL compilers. Second, the meta-program provides implementation for the `get_state` and `set_state` methods. This generated code is added to the analyzed class, which is also declared as implementing the `PSerializable` interface (for Portable Serializable).

The `PSerializable` interface is given in Figure 5. It defines the `get_state` and `set_state` methods and two additional operations: `get_Anystate` and `set_Anystate`. These operations are respectively devoted to the serialization (deserialization) of a CORBA object to (from) an `any`. Then, the supplied `any` is marshaled to a `State` buffer that is finally returned by `get_state`. The `set_state` operation is responsible for un-marshalling any variables from `State` buffers. These `anys` are then supplied to `set_Anystate`, which use `anys'` contents for updating object states. Figure 6, shows how the operations `get_state` and `set_state` are implemented. It is worth noting that thanks to the use of `any` variables and CDR buffers, this code is generic and can be used for handling any CORBA state container.

```
typedef sequence<octet> State;
interface PSerializable{
    State get_state();
    void set_state(in State st);
    any get_Anystate();
    void set_Anystate(in any st);
};
```

Figure 5. The `PSerializable` IDL interface

The rest of this section focuses on the problem of providing implementation of the `get_Anystate` and `set_Anystate` operations. For the former, we focus on the problem of saving an object state to a CORBA state container. For the latter, we tackle the opposite problem.

Table 1 specifies the code generated for handling serialization and de-serialization of built-in attributes, strings and CORBA object references. This table adopts two conventions: first, the CORBA state container used is called *StateContainer*; and second, the object attribute handled is named *a* and is accessed by *this.a*. As showed in the first row, built-in types are directly saved to and restored from their respective *StateContainer* fields. The second row is about string attributes, which are saved using the standard `CORBA::string_duplicate` method. Obviously, this duplication can be only performed when the string is not empty. The homologous and symmetric process is followed for de-serializing strings from state containers. Finally, the third row tackles CORBA object references. Remember that our mapping rules associates CORBA object references to IDL strings. Handling these references is the responsibility of the ORB that provides two methods: `object_to_string` and `string_to_object`. It must be noted that the de-serialization of an object reference is made in two steps:

first, the string is transformed by the ORB into a `CORBA::Object`, a common super-class for any object reference; second, that object is down-casted (narrowed) to its adequate type.

```
State PSerializable_impl::get_state() {

    // An any variable containing a CORBA state container
    // is retrieved here.
    CORBA::Any any = this.get_Anystate();

    // The any is then marshaled
    CDRBuffer buffer;
    buffer.Marshal(any.type());
    buffer.Marshal(any.value());

    // A CDR buffer is an abstract sequence of octets.
    // Hence, it can be saved in a State variable.
    State st = buffer;
    return st;
}

void PSerializable_impl::set_state( State st) {

    CDRBuffer buffer = st;

    // The any saved in the State variable is un-marshaled
    // from the CDR buffer
    CORBA::Any any;
    buffer.UnMarshal(any);

    // The obtained any contains the type and the value
    // of a CORBA state container. This contents is
    // interpreted by the set_Anystate operation
    this.set_Anystate(any);
}
```

Figure 6. Implementation of the `PSerializable` interface.

	Serialization	De-serialization
<b>Built-in type</b>	<code>StateContainer.a = this.a;</code>	<code>this.a = StateContainer.a;</code>
<b>String type</b>	if ( <code>a.length() &gt; 0</code> ) then <code>StateContainer.a = string_duplicate(this.a);</code> else <code>StateContainer.a = NULL;</code> endif	if ( <code>StateContainer.a != NULL</code> ) then <code>this.a = string_duplicate(StateContainer.a);</code> else <code>this.a = NULL;</code> endif
<b>CORBA Object reference type</b>	<code>StateContainer.a = obj_to_string(this.a);</code>	<code>CORBA::Object obj = string_to_object(StateContainer.a);</code> <code>this.a = CORBAObjRefType::_narrow(obj);</code>

Table 1. Serialization and De-serialization Generation rules (Built-in variables, Strings and CORBA Object References)

Table 2 follows the same conventions defined for table 1, but it concentrates on the serialization of class type attributes. In the implementation, a class type attribute is a composite object holding an object reference (identifier) that must be unique in a given implementation. The way of expressing this notion of object reference varies from one system to another. We express this concept in Table 2 using the general notation "`Ref(this.a)`".

In our approach, an object is serialized when its reference is not `NULL` and if it has not been already serialized. A table, called *serializedObjects*, contains the references of the objects already serialized. As stated in section 3.3.2, if an object has been already serialized, then additional attempts to serialize

its state results in saving only its reference (see first row of Table 2).

The de-serialization process (second row of Table 2) is performed according to the information supplied by the considered *StateContainer*. When a *Reference* identifies an object that has already been de-serialized, the current reference of this object can be retrieved from the *deSerializedObjects* hash-table. If this is not the case, this means that the object has not been already de-serialized. Then the reference is marked as “waiting for de-serialization” of this object. It must be noted that an object A having a reference “Ref<sub>1</sub>(A)” in one implementation will be identified through a different (new) object reference “Ref<sub>2</sub>(A)” in the context of another implementation. The *deSerializedObjects* hash-table maintains the correspondence between the object reference supplied by the CORBA state container (the container reference) and the current reference of the object (the one obtained when the object is restored). As a result, when an object is eventually de-serialized, its current reference and its container reference are stored in the *deSerializedObjects* hash-table. Then, every object reference marked as “waiting for de-serialization” of the container reference can be updated.

Serialization	De-serialization
<pre> ClassType classTypeObject; if (Ref(this.a) == NULL) then   classTypeObject = Reference(NULL); else if (SerializedObjects.HasReference( Ref(this.a) )) then   classTypeObject = Reference( Ref(this.a) ); else   classTypeObject =     ReferenceAndState( Ref(this.a), this.a.get_Anystate()); SerializedObject.PushReference( Ref(this.a) ); endif StateContainer.a = classTypeObject; </pre>	<pre> switch (StateContainer.a._discriminator){ case Reference:   if (StateContainer.a.ref == NULL) then     this.a = NULL;   else if ( deSerializedObjects.HasReferenceOf(StateContainer.a.ref)) then     this.a = deSerializedObjects.GetReferenceOf(StateContainer.a.ref);   else     deSerializedObjects.WaitingForDeSerialization(StateContainer.a.ref, Ref(this.a));   endif end; case ReferenceAndState:   if (this.a != NULL) then release(a); endif   this.a = new ClassType_of_a;   this.a.set_Anystate( StateContainer.a.state );   deSerializedObjects.PushContainerAndCurrentReference(StateContainer.a.ref,     Ref(this.a));   deSerializedObjects.UpdateWaitingForDeSerialization(StateContainer.a.ref); end; endswitch </pre>

**Table 2. Serialization and De-serialization rules (Class types)**

Constructed types (structures, arrays and sequences) are handled recursively. Thus, the rules presented in Table 1 and Table 2 are applied directly to each structure field and to each array or sequence element. Table 3 exemplifies this recursive approach for the case of a structure, an array and a sequence. In the first case, each field of the structure is sequentially saved and restored. In the second case, a loop is generated for iterating on array elements. As one can see, the body of the loop is defined using the mapping defined in table 1 on each built-in type element of the array. In the third case, sequences are handled following an incremental approach: each time that a new element is saved in the sequence, the length of this sequence is incremented by one. This results in the automatic allocation of the memory required by the new element. Conversely, if the length of a sequence is initialized to zero, then all the memory currently allocated for that sequence is automatically released.

	Serialization	De-serialization
Structure type	<pre> StateContainer.a.field1 = this.a.field1; ... StateContainer.a.fieldN =   this.a.fieldN </pre>	<pre> this.a.field1 =   StateContainer.a.field1; ... this.a.fieldN =   StateContainer.a.fieldN </pre>
Array type	<pre> for (i<sub>1</sub> [] [i<sub>1</sub><sup>min</sup> ... i<sub>1</sub><sup>max</sup>]       ...       i<sub>n</sub> [] [i<sub>n</sub><sup>min</sup> ... i<sub>n</sub><sup>max</sup>] )   StateContainer.a[i<sub>1</sub>...i<sub>n</sub>] =     this.a[i<sub>1</sub>...i<sub>n</sub>]; endfor </pre>	<pre> for (i<sub>1</sub> [] [i<sub>1</sub><sup>min</sup> ... i<sub>1</sub><sup>max</sup>]       ...       i<sub>n</sub> [] [i<sub>n</sub><sup>min</sup> ... i<sub>n</sub><sup>max</sup>] )   this.a[i<sub>1</sub>...i<sub>n</sub>] =     StateContainer.a[i<sub>1</sub>...i<sub>n</sub>]; endfor </pre>
Sequence type	<pre> StateContainer.a.length(0); for ( i = 0 to this.a.length() -1 )   StateContainer.a.length(i+1);   StateContainer.a[i] =     this.a[i]; endfor </pre>	<pre> this.a.length( 0 ); for ( i = 0 to   StateContainer.a.length()-1 )   this.a.length(i+1);   this.a[i] =StateContainer.a[i]; endfor </pre>

**Table 3. Serialization and De-serialization generation rules (Structures, Arrays and Sequences)**

As stated in section 3.3.1, we only enable the use of implementation simple inheritance, which is handled by recursion. Hence, the meta-program generates a recursive invocation to *get\_Anystate* or *set\_Anystate* for the super-class associated to each considered class type instance. These generation rules are defined in Table 4.

Serialization	De-serialization
<pre> if (this.HasSuperClass())   StateContainer.super =     super.get_Anystate(); </pre>	<pre> if (this.HasSuperClass())   super.set_Anystate(StateContainer.super); </pre>

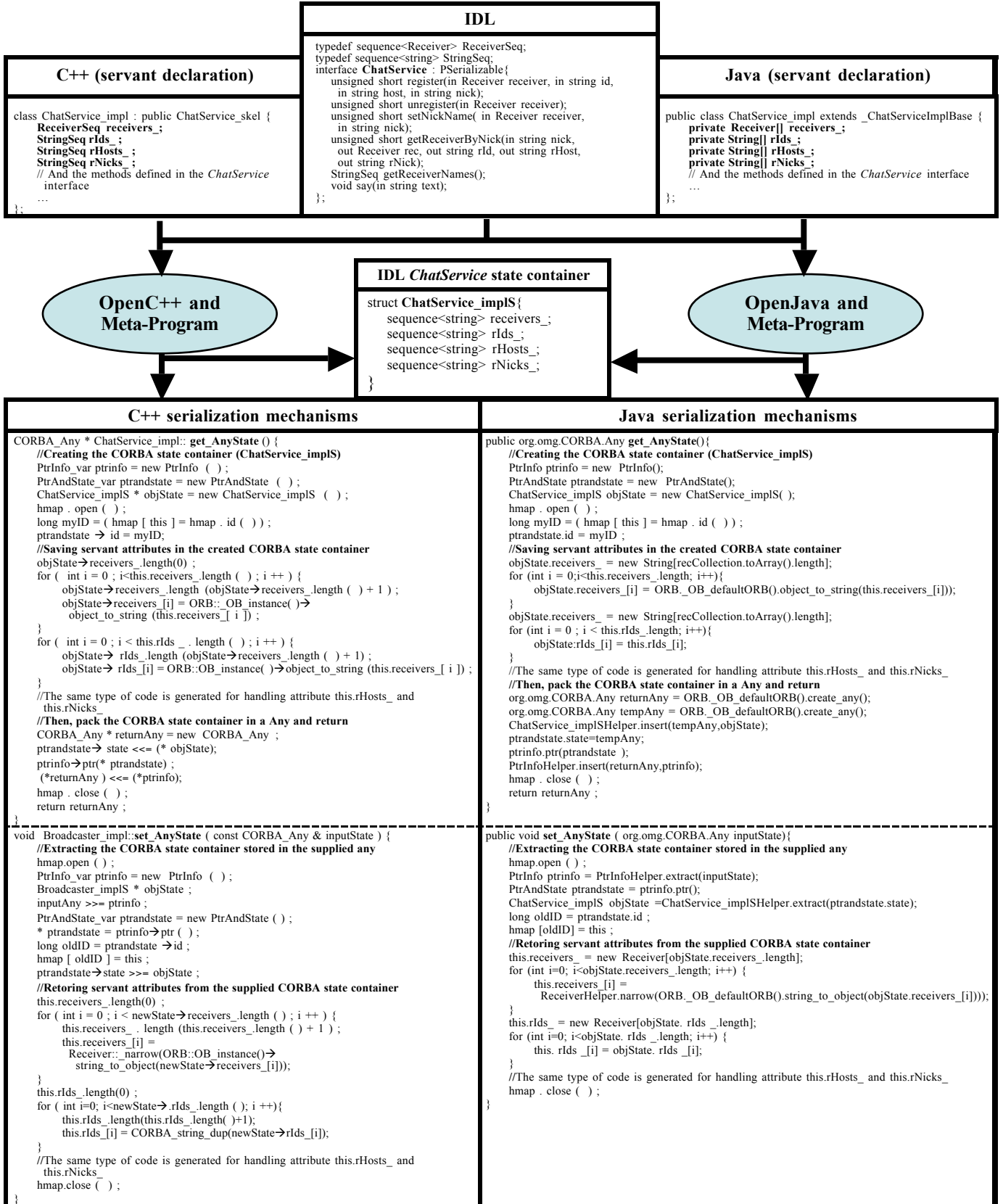
**Table 4. Serialization and De-serialization generation rules (Inheritance)**

One may consider that our approach obviates the handling of user-defined types. However, these types are essentially aliases to the types already presented. Thus, the rules defined in tables 1-4 are also valid for handling user-defined types. The meta-program must only determine the primitive type associated to a user-defined type and then apply accordingly the adequate generation rules.

## 4. CASE STUDY & LANGUAGE MAPPING

In this section, we illustrate the proposed technology using the example of a *Chat* server. This server is mapped onto a single system process, which is considered as an error confinement area<sup>1</sup>. The server has been designed in detail and implemented in both C++ and Java, respectively on top of Orbacus and JOrbacus 3.3.1. We show that these two implementations are interoperable at the state level. The example shows also that the conformance rules enforced by our approach are not too restrictive as they allow non- trivial states to be managed. Then, we discuss some language specific issues related to the mapping of the object state model to both C++ and Java.

<sup>1</sup> This implementation requirement is crucial in our context in order to limit the impact of a server failure over the rest of the system. Process boundaries provide an isolated address space ideal to handle this requirement.



**Figure 7. Chat Service Serialization**

## 4.1 Example

This section refines the example supplied in Section 3: the *ChatService*. This server implements a simple chat service where clients can: connect to, give their nickname, send messages to every client or to a particular client and obtain the list of the connected clients. Starting from the same design, we implemented both a C++ and a Java version of the service, the clients are written in C++ but could be also ported to Java or any other language. These two versions were compiled using our extended compilation facilities, and then have been extended with the PSerializable interface. Figure 7 shows the generated IDL *ChatService* state container and the C++/Java implementations of the *get\_AnyState* and *set\_AnyState* methods. One can note that the two implementations are pretty similar. The reason is simple, we use the same meta-program (the same analysis and transformation rules) in both cases; the only difference concerns the language and the open compiler (OpenC++ [6] and OpenJava [7]) respectively used to describe and apply the meta-program.

From a practical viewpoint, this example shows how the *ChatService\_impl* state container is packed to and unpacked from any variables. In C++, this is done by using the `<<=` and `>>=` operators. In Java, using *Helper* classes generated by the IDL compiler. Another practical issue concerns the use of local ORB instances inside CORBA object implementations. As stated in section 3.3.3, the stringification of object references involves the ORB. This justifies why most ORB providers supply to programmers a set of (proprietary) facilities to access to the ORB. In practice, there is typically only one ORB instance in each CORBA object. In Orbacus, this instance may be accessed by using the static ORB method `_ob_instance()`. In Jorbacus, the same is performed using the static ORB method `_OB_defaultORB()`. It is worth noting that from the serialization viewpoint, ORB instances are not part of the attribute facet but rather they belong to the platform facet.

### 4.1.1 An External State Observer

In order to illustrate the portability of the serialization provided, we implemented in Java an external state observer. This state observer is able to obtain the state of any CORBA object compiled using our technique. Figure 8 shows the graphic user interface of this observer: the user has to fill the *Object IOR* field (Interoperable Object Reference) and when the *Get Object State* button is pressed, the right panel shows a hierarchical view of the object's state. The state shown figure 8 is the state of the C++ chat server with two clients connected.

This observer can be used to debug an application, by observing and checking the state for conformance and it is a first illustration of the portability of the serialization process: the state of a C++ object is analyzed at runtime and displayed by a Java application.

### 4.1.2 Migration between C++ and Java Servers

Since we implemented two versions of the same chat server, both of them derived from the same detailed design, i.e. the C++ and Java classes have the same attributes. We set up an experience where the state of the servers are exchanged between each other, here is the scenario:

- The C++ server is started; two clients connect and exchange some greetings.
- The state of the C++ server is obtained and the server is crashed.

- The Java server is started and its state is restored from the state obtained at the previous step.
- The clients can continue to chat without any disturbance.

This scenario can be applied the other way around: the state is obtained from the Java server and applied to the C++ server.

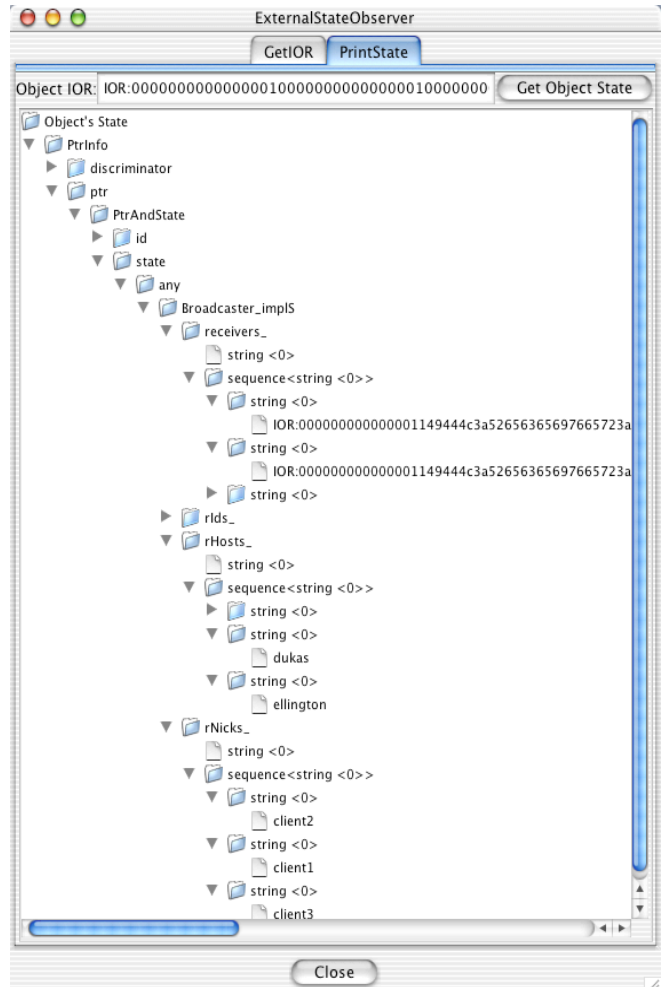


Figure 8. Interface of the Java External State Observer

### 4.1.3 Persistence of the Chat Server

Based on the external state observer, we implemented a Java application that obtains the state of a CORBA object and saves this state to disk. Later, the state can be read from the disk and applied to any chat server implementation to restore it (see figure 9).

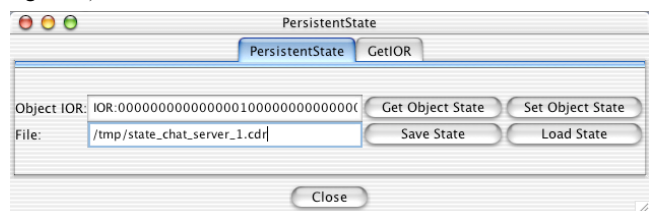


Figure 9. Interface of the Java Persistent State Server

Using this, the servers can be made persistent easily: each time a client connects or disconnects, the server can save its state to disk. When a server is started from scratch, it can restore its

state from the disk. Again, this persistence is applicable to C++ and Java servers interchangeably.

## 4.2 Language Mapping

It is very important to consider the practical restrictions resulting from the application of the defined object state model and its associated conformance rules. This is crucial for proposing programming alternatives for each defined restriction. Otherwise, the programming conventions imposed could be too restrictive.

In Java, there is no programming restriction since the language is conformant with the object state model we defined. A clear mapping between Java and IDL is defined in [12]. In C++ however, some restrictions must be obeyed (see Table 6).

Issue	C++
Multiple inheritance	Forbidden
Multi-level pointers <sup>2</sup>	Forbidden
Pointer arithmetic	To be forbidden if it can have side effects on other CORBA objects, i.e. if each CORBA object is not in a separated system process.
Unique IDL mapping	Some data types are ambiguous, like char types and thus forbidden (see discussion in section 3.3).

Table 6. Programming restrictions for C++

It is worth noting that the restrictions are very language dependent. Pure object oriented languages are clearly conformant with the state model we have defined. When it is not the case, then, some limited programming conventions must be observed. Although this may appear as a limitation, in particular when dealing with legacy applications, we believe that this is a way to enforce better object-oriented programming that is highly needed for dependability reasons. This is yet common practice in safety critical applications, e.g. in avionics.

Finally, the proposed approach could be extended relatively easily to other programming languages, object-oriented or not, provided that an open-compiler and a mapping are available, but this is beyond the scope of this paper.

## 5. APPLICATION AND DISCUSSION

This technology can be applied to a number of domains/applications: mobile agents, dependable computing, load balancing, etc. We illustrate here a possible use of this technique in some of these domains.

### 5.1 Mobile Agents

Being able to serialize objects in a language independent way can be very interesting in the context of mobile agents. A simple definition of what are mobile agents is: small objects that can travel the network in order to realize one or several tasks at different *locations* (hosts). A typical application is the

<sup>2</sup> Multi-level pointers are pointers expressing more than one level of indirection in the access to a variable, as `int**`, `int***`, and so on. Simple-level pointers, like `int*` or class pointers are allowed.

travel-agent: the agent is set up at *home* so that it will scan a list of different car rentals (or airlines, hotels, etc.) on behalf of the user to search for an offer matching some given criterions and to minimize (or maximize) a given function, e.g. usually it will minimize the price. A typical problem with this scenario is that it requires the different car rental companies to have the same agent-hosting platform (system/middleware and language or virtual machine) running on their network. This is far from being the case currently and we doubt that it will in the near future.

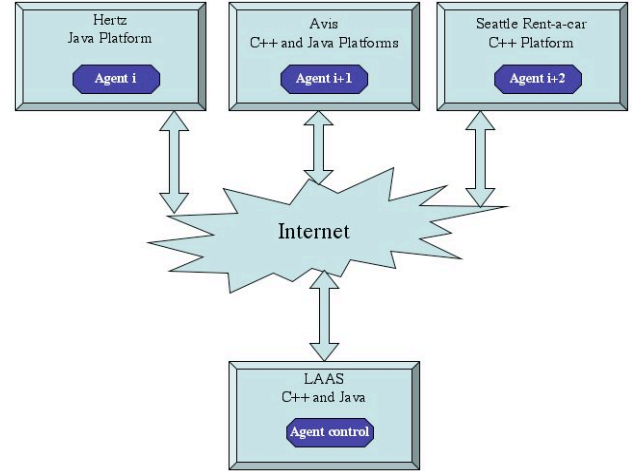


Figure 10. Example of mobile agent migration.

Using our approach, we can have several implementations of an agent derived from the same design. The serialized agent can be used by any implementation and then the above problem is solved: the agent can be moved from home to Hertz, incarnated in Java, serialized and traveled to Avis, incarnated in C++, etc. until it comes back home with the results. This is a very simple scenario but it illustrates that portable serialization is a basic support technology for ubiquitous computing.

It is worth noting that for mobile agents, handling the attribute facet is sufficient because these objects are aware of mobility, they thus don't use platform dependent attributes and they manage their own communication.

### 5.2 Checkpointing and dependability

Distributed systems provide a convenient (and natural) framework for the development of replication-based fault-tolerant strategies [13]. One of the major issues in the design and implementation of distributed fault-tolerant systems is checkpointing. *Checkpointing* can be simply defined as the process of saving a program state so that it may be restored later in time. Checkpointing a distributed application involves complex algorithms to ensure the consistency of the distributed recovery state of the application. In particular, handling global checkpoints requires, among other things, getting a correct set of local checkpoints for the entities (objects in object-oriented systems) participating in the distributed computation.

In addition, basic distributed fault tolerance mechanisms rely on replication. Checkpoints reflecting the current state of each individual object are needed for replica synchronization and for cloning replicas. Hence, a basic brick for the development of distributed replication mechanisms is obtaining local object states.

Most available checkpointing techniques rely on memory snapshots. This approach assumes that replicas are strictly identical and are de-facto running on the same platform, which is often the case. In this approach, there is no semantics attached to the checkpointed information. This raw information is used as is to update the state of backup replicas. This approach often assumes a limited fault model, essentially only crash faults are considered. Triplication and voting considers more subtle faults. However as replicas are identical only physical faults can be handled with this technique.

The proposed technique provides more meta-information and a portable format that enables replicas to be developed in different programming languages from a same detailed design. For several decades, it is well known that software faults are most likely to occur in today's system [14]. Although this technique cannot help very much for tolerating design faults at the object level, since the same detailed design may lead to common mode failure, it is of high interest when considering design or implementation faults at the underlying software platform level. Temporal redundancy and software rejuvenation [15] are the sort of techniques able to tolerate this kind of faults. Many of these faults are related to the aging of systems in operation. Although more work is needed to this aim, we believe that this technique can be of high interest to deal with software faults at the system platform level.

Roll-back mechanisms can also take benefit of a portable serialization technique, in particular in a CORBA context, for large scale transactional systems.

Another interesting issue is the possibility of implementing, based on the technique proposed in this paper, tools for managing the evolution of software. One could easily imagine a customized version of the deserialization mechanism that would be able to interpret and to perform some basic modifications on the serialized state before the proper deserialization. These modifications could include, for instance, the translation of a *short integer* into a *float* provided that the corresponding attributes in the old and in the new version of the object have the same name.

### 5.3 Discussion

The above sections illustrate the interest of a portable serialization technique. Many more examples can be found, e.g. load balancing, GRID computing, supervision of large applications, distributed debugging, etc. We believe that open compilation and related techniques, such as AOP, are very promising techniques to master complexity of application object-oriented programming and to tune the implementation according to the system requirements. Understanding better the state of objects is of high interest to adjust the implementation according to several objectives. It is worth noting that being able to trace the state of CORBA objects has many benefits for testing reasons. From a performance viewpoint too, this gives interesting insights on the behavior of the object at runtime. These feedbacks can be used to optimize the organization of the object attributes, the inheritance hierarchy, the object decomposition and the like. The visibility of generated mechanisms by the user enable checking the implementation details of a CORBA object. In addition, for optimization reasons for instance, the user is able to modify the automatically generated methods.

However, it is clear that in many systems the attribute facet is not sufficient to handle object states. This is mainly due the

fact that a CORBA system is not a pure object-oriented system. Rather, it is a hybrid system, grouping both object-oriented and traditional imperative programming. Any system call performed by an object to the operating system has two effects: (i) the call creates or updates internal data structures within the operating system address space and (ii) returns local identifiers which are only valid on a single site. This local value returned to the object is stored in the object address space in a site-dependent variable. This information is thus not valid on a different site and thus must not be checkpointed as is. This means that more meta-information is needed to handle this type of information. This involves additional mechanisms (i) to identify this type of variable, (ii) intercept system calls like this, (iii) journalize all system calls performed by an object and (iv) perform the system calls at the destination site to re-create or update the system platform and get the correct value of the site-dependent variables on the remote site. This kind of mechanisms is quite conventional in fault tolerant computing and transactional systems. As far as hybrid systems are considered, then, they are means to handle part of the platform facet, related to explicit system calls.

However, in particular in middleware systems such as CORBA, many actions performed by the middleware are not related to explicit calls from the application object. The interface of a middleware such as CORBA is rather implicit (ORB and POA). Few ORB services belong to an explicit API (some initialization and declaration routines). This is why the internal state of a middleware such as CORBA cannot be handled in a way as described above.

The most promising approach relies on using other reflective layers like reflective middleware [16] and reflective operating systems [17] to externalize useful state information. This means that reflective components can be setup to make some part of their internal state visible or even adjustable on a case-by-case basis, depending on the aims of the system designers. This approach has many merits since, beyond solving in an elegant way the problem of the platform facet, it enables tuning the tradeoff between usefulness and runtime overheads. Such a multi-level reflective approach has also many merits regarding the communication facet since it can externalize the state of protocol stacks. Interception facilities at communication level enable message logging and end-to-end protocols to be implemented easily.

Clearly, reflective component technology, including middleware and operating system components is a very attractive field of investigation and a real challenge for future research. As far as state information is concerned, the identification of the components persistent state (e.g. open file descriptors) and the component volatile state (unnecessary information for recovery/restart) is something that must be looked at carefully. This has a direct impact on the definition of the needed reflective features, i.e. the definition of the minimal metainterface required to strictly access the necessary and sufficient component state information. Such technology is very promising to checkpoint threading, ORB internal variables and message queues.

## 6. RELATED WORK

### 6.1 Memory management approaches

Object serialization is nowadays a problem with particular relevance to fault-tolerance but also to mobile computing, object migration, adaptive computing and load balancing.

Memory management approaches are based on memory snapshots and control of memory allocation [18]. Processes own persistent memory regions and thus all objects allocated in these regions are persistent. However, these solutions do not provide the required granularity for serializing individual object states.

For instance, in the GUARDS System [2] developed for a wide range of application domains (e.g. railways, aerospace, etc.) state variables are declared a priori and mapped onto specific memory regions. Memory regions are in turn divided into blocks. Blocks may hold several state variables and are transferred to restore the state of the computation on a companion processor. A quite interesting algorithm enables a block to be transferred as soon as a state variable has been updated within this block. In this solution no semantic and type information is attached to the transferred information, since replication is performed on identical processors loaded with the same executive software.

This is why this type of solution cannot be used in a language heterogeneous environment.

## 6.2 VM and compiler-assisted solutions

We address in this section two sorts of approaches: those based on virtual machines (VMs) and those assisted by compilers. Solutions based on VMs [19] [20] benefit from an integrated runtime type support for object serialization that minimizes the required contribution from programmers. Indeed, the runtime of a language hold all the necessary information to get and restore object state information. This is why object persistence can be achieved with this approach in a transparent way. In Java for instance, the Java Reflection API enables the complete type /class information to be accessed at runtime. This information has however only meaning on the same virtual machine elsewhere.

Compiler-assisted solutions, and particularly those based on open compilers, have also shown their interest for providing transparent object checkpointing in systems without runtime support for serialization, like those based on C++ [21] [22]. The approach in this case is also based on open compilers but resulting state information is language dependent.

Another technique relies on source-to-source compilation to produce instrumented programs that are able to save state information in a binary-compatible format [23]. In Preaches [3] a portable checkpointing solution for single process applications is proposed. Instead of using an architecture-independent format, this approach generates machine-dependent checkpoints for each one of the considered architectures. A compiler-assisted solution very closely related to our work, is the one presented in [24]. This solution proposes a tool based on the JavaCC preprocessor in order to automatically generate state transfer code for CORBA applications. However, the approach only supports state transfers between object replicas written in the same language (C++ or Java).

Both types of solutions follow the same principle: they provide serialization of individual objects by analyzing and interpreting their structure dynamically (in the case of VMs), or statically (in the case of compilers). Despite their benefits, the application of these solutions to CORBA, DCOM [25] or Java/RMI [26] systems can be criticized since they impose the use of particular platforms and programming languages.

The solutions proposed by DirectToSOM (DTS) [30] and the Common Language Runtime of .NET (CLR) [31] are hybrid of the two types described above. They rely on the use of specific compiler on dedicated platforms to support the sharing of in-memory representation of an object at runtime. Although the goal is not exactly the same, these solutions seem quite interesting and we look forward reading more about them.

## 6.3 FT-CORBA

FT-CORBA extends CORBA with a set of IDL interfaces for the management of consistently replicated fault-tolerant objects. One of the major goals of this infrastructure is the provision of strong replica consistency when using active, warm and cold passive replication mechanisms. This requirement depends on the capacity of the fault-tolerance support for serializing object replicas. For active replication, each server replica computes and responds to every operation. When an active replica is recovered from a failure, its state must be synchronized with the state of the active replicas. For passive replication, one of the replicas, the primary replica, responds to every operation. With warm passive replication, the state of the remaining replicas, known as backups, is periodically synchronized with the primary replica's state. With cold replication, a backup replica is loaded into memory and its state is initialized from a log only when the existing primary replica fails.

This need of object serialization leads to the notion of CORBA *Checkpointable* object (a notion similar to the one of Portable Serializable object used in this paper). FT-CORBA states that every replicated CORBA object must inherit from the IDL *Checkpointable* interface. This interface provides two basic operations used for serializing (*get\_state*) and deserializing (*set\_state*) the state of an object. Because it is not possible to anticipate the format of the state of every object, the *State* of an object is defined as a sequence (stream) of octets. The implementation of these operations is left open to the application programmer.

DOORS [27] and Eternal [28] provide (partial) implementations of the FT-CORBA infrastructure. DOORS focusses on the definition of an architectural solution for an efficient implementation of FT-CORBA. Eternal provides a complete framework for the implementation of fault tolerant applications on top of CORBA. The merits of Eternal are two-fold: (i) the proposed infrastructure enables off-the-shelf ORB to be used thanks to the notion of interceptors, and (ii) the framework is consistent with the development of crash-fault tolerant systems. In this context, the implementation of *get-state* and *set-state* operation can rely on simple raw information checkpointing since replicas are running on identical software/hardware support.

## 6.4 Other OMG efforts

The OMG has recently adopted the definition of a new generic CORBA service called the *Persistent State* service [29]. The basic idea is to introduce a new language in CORBA, called PSDL (Persistent State Description Language), for the declaration of object states. Among other features, this language will provide built-in support for declaring object states and the resulting PSDL definitions will map to the most common programming language. We think that this approach is quite promising, especially for the provision of language and platform independent object state repositories. Unfortunately, programmers are responsible for providing

such PSDL declarations. The technique we proposed in this paper should enable PSDL declaration to be automatically generated from object implementations. This is a track for future work.

## 7. CONCLUSION

We have shown in this paper the interest of reflection and open compilers to the handling of object state internal information. We applied this type of technique to CORBA objects and took advantage of CORBA features to define a portable format of state information. We have shown that many application fields and techniques can benefit very much from this approach. Among other fields, mobile computing over CORBA platforms and fault tolerant computing through portable checkpointing facilities can be improved very much.

Our contribution is two-fold. We have developed a state generator tool that enables state information (attribute facet) and save/restore methods. This tool was applied to a case study showing the interest of the technique to exchange state information between CORBA servers developed with different object-oriented programming languages. In addition, a state observer tool was also developed and used to visualize the state of these CORBA servers in operation.

This work is consistent and complementary with OMG efforts, including FT-CORBA and PSS. The latest developments on these topics (e.g. PSDL) are driving forces promoting the notion of portable state information. In addition, they provide new inputs to extend the work presented in this paper.

Reflection is a very attractive concept as it supports the notion of separation of concerns. This concept will enable other facets of the object state in a hybrid system to be handled. More work is needed to tackle all the facets of the problem. However, recent initiatives in both reflective middleware and operating systems are really promising. A clever combination of conventional interception techniques and reflective component technology should enable both platform and communication facets to be addressed in a very elegant way. More experimental work is needed to assess this type of technique in several fields and to evaluate the necessary tradeoffs between efficiency and performance.

## 8. Acknowledgements

This work was partially supported by the European Community (project IST-1999-11585: DSOS – Dependable Systems Of Systems).

## 9. REFERENCES

- [1] Plank, J.S., M. Beck, and G. Kingsley, "Compiler-Assisted Memory Exclusion for Fast Checkpointing", in *IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance*. 1995.
- [2] Powell, D., *A Generic Fault Tolerant Architecture for Real-Time Dependable Systems*. 2001: Kluwer Academic Publishers. 242 pages, ISBN: 0-7923-7295-6.
- [3] Ssu, K.-F. and W.K. Fuchs. "PREACHES - Portable Recovery and Checkpointing in Heterogeneous Systems" in *28th IEEE Fault-Tolerant Computing Symposium*. 1998. Munich (Germany). pp. 38-47.
- [4] Strumpen, V. and B. Ramkumar, "Portable Checkpointing for Heterogeneous Architectures". *Fault-Tolerant Parallel and Distributed Systems*, Avresky and Keli Eds. Kluwer Academic Press, 1998: p. 73-92.
- [5] Maes, P. "Concepts and Experiments in Computational Reflection" in *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*. 1987. pp. 147-155.
- [6] Chiba, S. "Macro processing in object-oriented languages" in *Technology of Object-Oriented Languages and Systems (TOOLS'98)*. November 1998. Australia. pp. 113-126.
- [7] Tatsubori, M., et al., *OpenJava: A Class-based Macro System for Java*, in *Reflection and Software Engineering*, Springer Verlag, Editor. 2000, LNCS 1826. pp. 119-135.
- [8] Kiczales, G., et al. "Aspect-Oriented Programming" in *European Conference on Object-Oriented Programming (ECOOP'97)*. 1997. Jyväskylä, Finland. pp. 220--242.
- [9] Kiczales, G., J.d. Rivières, and D.G. Bobrow, *The Art of the MetaObject Protocol*. 1992, Cambridge: The MIT Press. 335 pages, ISBN: 0-262-61074-4.
- [10] Object Management Group, *CORBA 2.5 specification*, <http://www.omg.org/cgi-bin/doc?formal/01-09-01>, 2001.
- [11] Object Management Group, *C++ Language Mapping Specification*, <http://www.omg.org/cgi-bin/doc?formal/99-07-41>, 1999.
- [12] Object Management Group, *Java Language Mapping to OMG IDL*, [http://www.omg.org/technology/documents/formal/java\\_language\\_mapping\\_to\\_omg\\_idl.htm](http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm), 2001.
- [13] Powell, D. "Distributed Fault-Tolerance: A Short Tutorial" in *IFIP International Workshop on Dependable Computing and its Applications (DCIA'98)*. 1998. Johannesburg (South Africa). pp. 1-12.
- [14] Gray, J. "Why do computers fail and what can be don about it?" in *International Symposium on Reliable Distributed Systems*. 1986. Los Angeles, CA (USA). pp. 3-12.
- [15] Huang, Y., et al. "Software rejuvenation: Analysis, Module and Applications" in *25th Int. Symposium on Fault Tolerant Computing*. 1995. Pasadena, CA (USA). pp. 381-390.
- [16] Costa, F.M., G.S. Blair, and G. Coulson, "Experiments with an architecture for reflective middleware", in *Integrated Computer-Aided Engineering*, IOS Press. 2000.
- [17] Yokote, Y. "The Apertos Reflective Operating System: The Concept and Its Implementation" in *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*. 1992. pp. 414-434.
- [18] Singhal, V., S. Kakkad, and P. Wilson. "Texas: An Efficient, Portable Persistent Store" in *5th International Workshop on Persistent Object Systems*. 1992. S. Miniato (Italy) pp.11-33.
- [19] Lutz, M. and D. Ascher, *Learning Python*. 1999. 382 pages, ISBN: 1-56592-464-9.

- [20] Sun, *Java Object Serialization Specification - Release 1.2*, <ftp://ftp.java.sun.com/docs/j2se1.3/serial-spec.ps>, 1996.
- [21] Ruiz, J.C., et al. "Optimized Object State Checkpointing using Compile-Time Reflection" in *IEEE Workshop on Embedded Fault Tolerant Systems (EFTS'98)*. 1998. Boston, USA. pp. 46-48.
- [22] Kasbekar, M., et al. "Issues in the design of a reflective library for checkpointing C++ objects" in *18th IEEE Symposium on Reliable Distributed Systems (SRDS'99)*. 1999. Lausanne, Switzerland. pp. 224-233.
- [23] Strumpen, V., "Portable and fault-tolerant software systems", in *IEEE Micro*. Septembre/Octobre, 1998. pp. 22-32.
- [24] Tewksbury, L.A., L.E. Moser, and P.M. Melliar-Smith. "Automatically-Generated State Transfer and Conversion Code to Facilitate Software Upgrades" in *International Conference on Software Maintenance*, 2001.
- [25] Thai, T.L., *Learning DCOM*, ed. A. Oram. 1999, Sebastopol, CA 95472 O'Reilly & Associates, pages, ISBN 1-56592-581-5.
- [26] Sun, *Java™ Remote Method Invocation (RMI)*, <http://java.sun.com/j2se/1.3/docs/guide/rmi>, 1999.
- [27] Natarajan, B., et al. "DOORS: Towards High-performance Fault-Tolerant CORBA" in *2nd International Symposium on Distributed Objects and Applications (DOA 2000)*. 2000. Antwerp, Belgium. pp. 196-204.
- [28] Narasimhan, P., L.E. Moser, and P.M. Melliar-Smith. "State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects" in *International Conference on Dependable Systems and Networks (DSN'01)*. 2001. Goteborg (Sweden). pp. 261-270.
- [29] Object Management Group, *Persistent State Service 2.0*, <http://www.omg.org/cgi-bin/doc?orbos/99-07-07>, 1999. Formally adopted in October 2001.
- [30] J. Hamilton, *Programming with DirectToSOM C++*, John Wiley & Sons, 1996.
- [31] E. Meijer and J. Gough, *A Technical Overview of the Common Language Infrastructure*, Microsoft Research. <http://research.microsoft.com/~emeijer/Papers/CLR.pdf>