

# OVIBUS: A Scalable Platform for Combining Heterogeneous Components

Francisco Domínguez, Rafael Capilla<sup>1</sup>, Juan Manuel Cueva<sup>2</sup>

<sup>1</sup> Departamento de Ciencias Experimentales y Tecnología,  
Universidad Rey Juan Carlos, c/ Tulipán s/n,  
28933 Madrid, Spain

{f.dominguez, rcapilla}@escet.urjc.es

<sup>2</sup> Departamento de Informática, Universidad de Oviedo, Avda. Calvo Sotelo s/n,  
33007, Oviedo, Spain

{cueva}@lsi.uniovi.es

**Abstract.** Component-based software development processes have achieved a great importance in the construction of software systems. Moreover, the development of applications using reusable components can reduce time-to-market. One of the main problems for developing software components is how they can inter-operate in different and heterogeneous platforms or systems. To solve this problem, several solutions have been proposed. From our point of view we believe that common platforms that make transparent the interoperability of components are necessary. In this work we outline a platform so-called OVIBUS that makes possible components interoperability between different technologies, such as: CCM, COM+, EJB, etc., in order to improve component-based software development.

## 1 Introduction

One of the most promising ways for building software systems is employing software components [14] as building blocks but the existence of different component technologies makes difficult the integration of such components. Several component technologies and platforms have been proposed to make possible the interoperability between different types of component. In this way, Corba components model (CCM), Sun's JavaBeans, Microsoft's COM+ [6] and more are examples of different technologies used today for building software systems. Nevertheless, all of these technologies have a low degree of heterogeneous scalability because all of these platforms have been designed to inter-operate in a homogeneous environment.

In addition to this, if we want to employ reusable components based on other technologies, we must solve how these components can be used by the platforms mentioned above. In this way, several solutions such as wrappers, bridges, adapters or connectors can be used to solve the integration problems.

In this work we propose a solution able to solve some of these problems but before this we have to mention some additional aspects related to the heterogeneous interoperability between some of the existing platforms, such as we describe in next section.

## **2 Heterogeneous Interoperability Problems**

If we want to make possible a heterogeneous interoperability between components belonging to different platforms, we will need to solve several problems, such as integration, communication, introspection, type conversion, parameter adaptation, etc. Related to this, some proposed solutions solve the problems mentioned above but only between two existing platforms, such as: CORBA [13] with COM (CORBA-COM OMG interoperability specification), Java with CORBA or EJB with CCM (CCM OMG specification), COM with Haskell [10]. As we can see, such solutions are too rigid because they don't consider the communication or interoperability with more than two platforms in a transparent mode for the user. Some of the interoperability problems for which we want to provide an appropriate solution are the following:

### **2.1 Wrapping Problems**

When we want to communicate two objects or components belonging to different platforms, one way to do this is using a wrapper [8]. A wrapper is a software item (e.g.: an object, a software component, a library, etc.) able to communicate with another item belonging to a different system. This wrapper offers the same services and resources offered by the original item (from now on we will use the term object instead of item because we use objects to communicate different components).

If we want to build a wrapper object we can do this in three different ways (as we know):

- ❑ The first way to build a wrapper is to build it manually. A software engineer (e.g.: a programmer) builds the wrapper manually each time is needed.
- ❑ The second way is using a bridge. A bridge is a program that facilitates the interoperability between platforms. Then, the bridge builds a wrapper for each component specified by the programmer. This process is carried out manually.

- The third way is build the wrapper through an automatic process. In this case, the system uses specific classes (we call to this a “wrapper factory”) for building a particular wrapper. This process is carried out without the intervention of the programmer.

The first and the second forms to build a wrapper program are the most used ones but they are not enough flexible because they don't allow to add new objects dynamically. The third way to build a wrapper seems to be the most powerful and flexible one but from our knowledge we haven't seen interoperability between different platforms using dynamic wrappers. We have only seen dynamic wrappers but only applied in the same platform [11].

## 2.2 Communication between Heterogeneous Objects

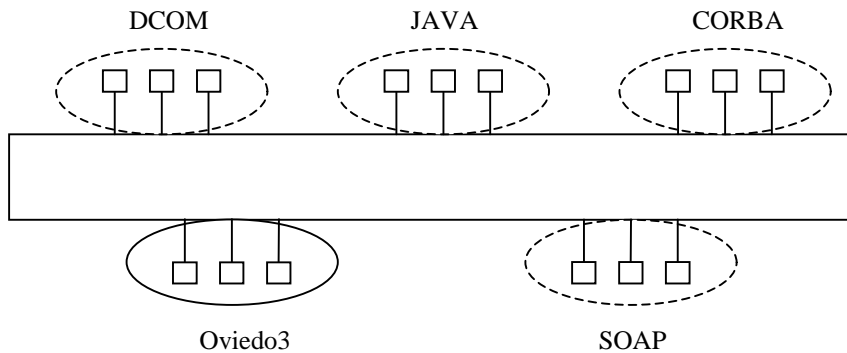
When we want to communicate heterogeneous object among different platforms several problems arise in the communication problem. Some of these problems are the following ones:

- **Class Naming:** this refers to the different ways to name classes by different platforms because this can lead to an incompatibility situation when two objects from different platforms want to use the services from the other platform. This problem occurs because each platform has a different naming service.
- **Dynamic Method Invocation:** this problem refers when we want to invoke objects belonging to different platforms. Each platform uses a particular way to invoke methods from the objects contained in the platform. Then, if we want to achieve a successful degree of interoperability we need to provide common invocation methods.
- **Introspection:** this refers to the ability of an object to observe and look inside the classes of another object or even itself. Introspection is an aspect of *reflection* (“is the ability of a program to manipulate as data something representing the state of the program during its own execution”) [2].

To solve some of the mentioned problems several platforms implements their own solutions. For instance, interoperability between Java and CORBA uses heritage for building the wrappers needed manually. Also, neither introspection nor dynamic invocation is needed because they are performed in compilation time, so the programmers need to know the name of the methods. Finally, Java has to use the CORBA Interface Repository to know the name of the classes in order to reference CORBA objects. In order to make more interoperable the variety of platforms, many of the services needed by the communication processes should be unified. In this work, we have tried to improve such problems by proposing an architecture that permits the communication of components among different platforms, such as we explain in next section.

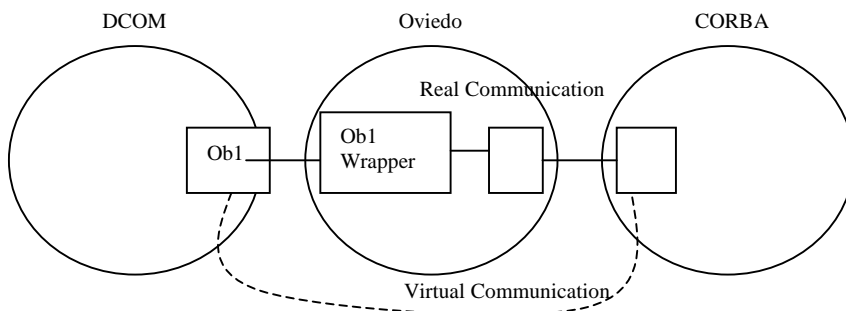
### 3 OVIBUS: A Platform for Component Interoperability

In this work we have proposed a scalable and heterogeneous architecture, so-called OVIBUS, to allow the interoperability between components of different platforms. Our main goal is to communicate different components from one platform without having knowledge about the other platforms. For instance, a CORBA programmer who wants to use COM objects he / she wouldn't need to know anything about COM. This will permit a transparent communication process among several platforms. In figure 1 we can see a global scheme of how the OVIBUS interconnects several platforms to permit the communication between different objects.



**Fig. 1.** The OVIBUS platform for interconnecting different components. The dotted circles represent system that can be plugged to the OVIBUS system. In the figure, Oviedo3 represents the core of the system.

To achieve the proposed interoperability described in figure 1, OVIBUS uses a dynamic wrapping method to enable a virtual communication channel between heterogeneous objects, such as is shown in figure 2.



**Fig. 2.** Interoperability process between CORBA and DCOM provided by the wrapper object.

As we see in figure 2, we establish a virtual channel between CORBA and DCOM objects through the wrapper object that are implemented dynamically by an object-oriented integral system, so-called Oviedo3 [1]. To see this wrapping process with more detail, we need to introduce some concepts describing how Oviedo3 [1] is implemented.

### 3.1 OVIEDO3: An Object-Oriented Integral System

The Oviedo3 [1] system is composed by two subsystems: an object-oriented abstract machine so-called Carbayonia and an object-oriented operating system so-called SO4.

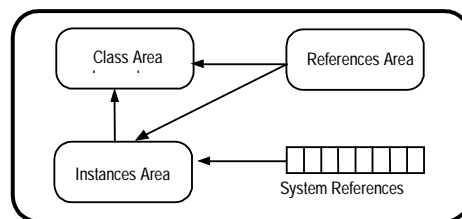
$$\text{Oviedo3} = \text{Carabayonia} + \text{SO4}$$

Carbayonia is a high level abstract machine<sup>1</sup> that provides a basic object environment upon the Oviedo3 [1] is built. The machine supports an object model with the following features: object identity and abstraction, encapsulation, inheritance and generic and aggregation relationships between objects and polymorphism. Also we have to mention, that the main characteristic of Carbayonia is that we manipulate objects through references to that objects.

The architecture of the Carbayonia abstract machine is composed of four parts so-called *areas* (each area is considered as a singleton object [Gam1995]), which are:

1. Class area. Maintains the description of each class.
2. Reference area. Stores the references of any existing object.
3. Instance area. Stores the created objects.
4. System reference area. Contains the references with specific functions in the machine.

In figure 3 we can see the four areas that form the Carbayonia machine.



**Fig. 3.** Carbayonia machine architecture.

---

<sup>1</sup> A high level abstract machine works directly with objects.

Some of the preliminary advantages derived from using an abstract machine are:

1. Portability.
2. Everything is an object, so we can remove the existing impedance mismatch between the applications, the operating system and Carbayonia.
3. Programming language independence.
4. Implementation of Operating System features.

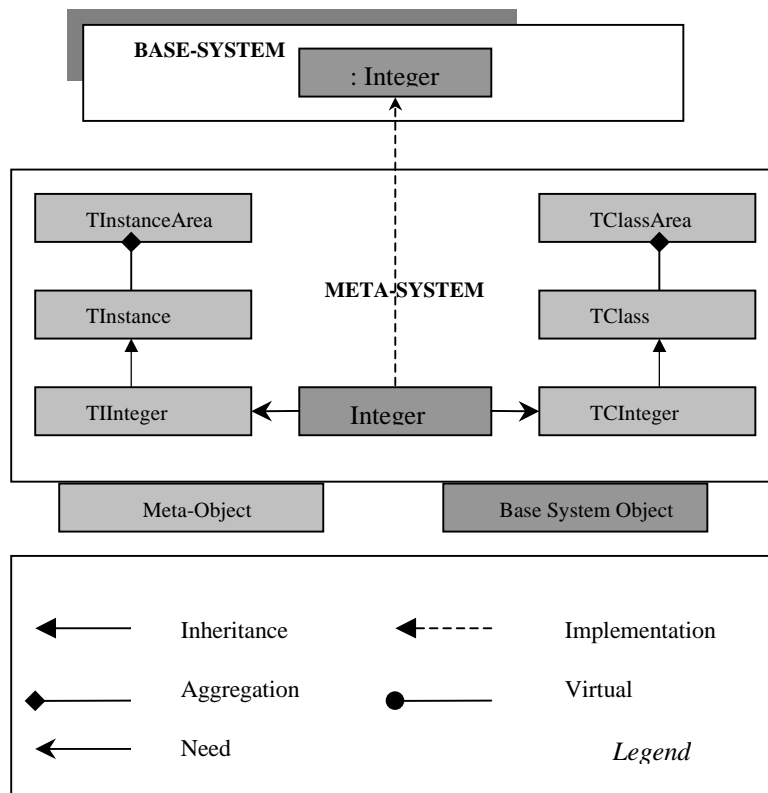
For the purpose of this paper, we don't consider necessary to describe the second part of the Oviedo3 system, that is the SO4 system.

### **3.2 A Dynamic Wrapping Process for Heterogeneous Interoperability**

To solve the interoperability problems between heterogeneous platforms and using the OVIBUS with the Oviedo3 platform, we describe in this section how the wrapping process is carried out. Our process provides a mechanism to generate dynamically wrappers [7] applied to components in a given platform. For a new platform we would need to repeat the same process. First we need to introduce some concepts.

Usually, a computational system is arranged in several levels but from the reflective architecture viewpoint are organized in two levels, which are: *the base-system level* which can be understood as a runtime level and a *meta-system* level that provides the infrastructure necessary to interpret the code of the base-system level. In our case, the meta-system is a simulator of the Carbayonia machine.

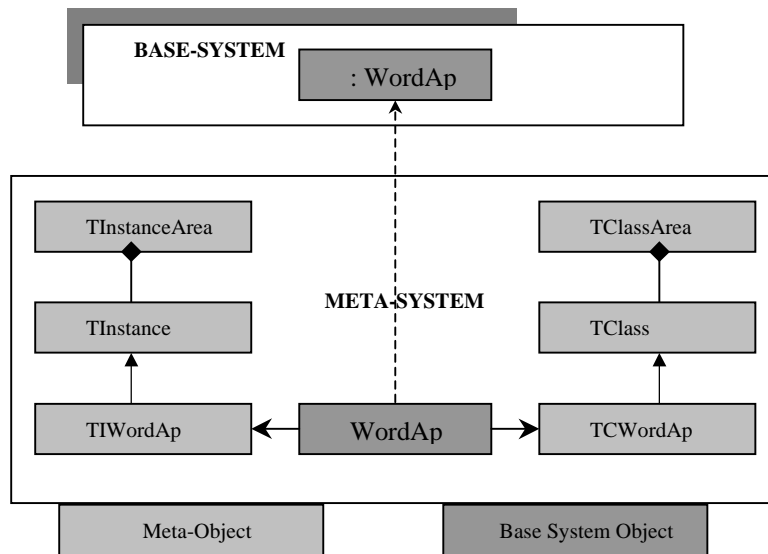
In figure 4 we can see an example of both levels describing how an integer number is described in the Carbayonia system.



**Fig. 4.** The Integer class implementation in the Carbayonia architecture.

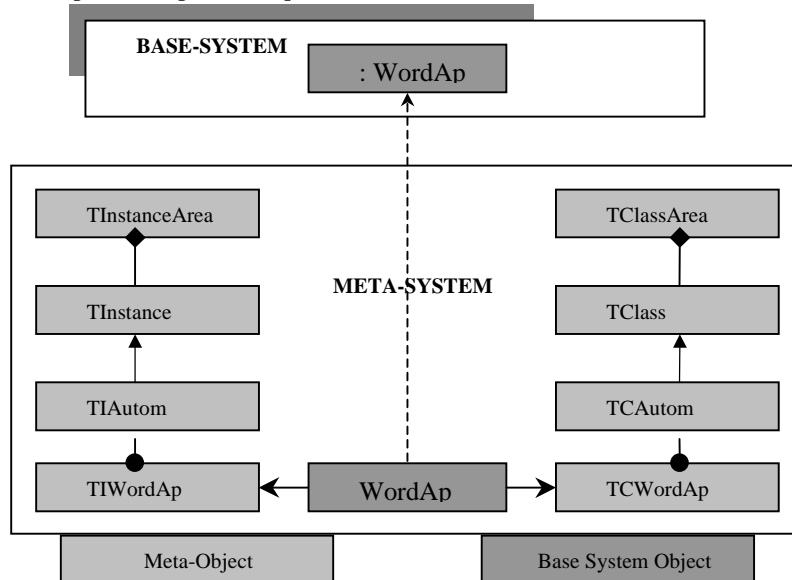
Each object in the base-system is implemented using two classes: TI\* (instance) and TC\* (class). Both classes are derived from TInstance and Tclass respectively. In this case, we have used TCInteger to implement the integer class and TInteger to implement, as many instances of the Integer class are needed.

In figure 4, we have defined the *need* relationship to indicate that the Integer class needs the existence of the classes associated to it. For example, if we want to implement the Integer class, we need TInteger and TCInteger. In addition to this, the implementation relationship provides visibility of a particular class in the base-system level. The need relationship specifies how the wrapper for the Integer class is implemented using both TInteger and TCInteger classes. Usually one unique class composes a wrapper, but in our proposal the wrapper is formed by the two classes mentioned before. This model allows us to interoperability between different platforms. For example, if we want to implement the Microsoft Word.Application object in the COM+ platform, we need to repeat the process described in figure 4 for that new object, such as we can see in figure 5.



**Fig. 5.** Implementation of the Word.Application component.

Figure 5 shows the usual way in which a wrapper is implemented manually. The problem arises if we want to extend this to the rest of the objects. This would be a very time and resources consuming task. To solve this problem we propose a solution that makes possible the implementation of the wrapper dynamically and we can apply this to all the components of a particular platform that need such solution. For example, for the case of COM+ objects such as can we see in the figure given below, we have developed the TCAutom and TIAutom classes that dynamically create the instances needed to wrap the component required.



**Fig. 6.** Dynamic Wrapping process for the Word.Application component.

In fact, the behavior of the instances of the classes described in figure 5 for the `Word.Application` component is equal to the behavior of the instances of the classes described in figure 6, that is: TIWordAp and TCWordAp.

To show how the user (i.e.: a programmer) can easily employ components from different platforms in a transparent mode, the following code shows an example of how the `word.application` component is used by the `Oviedo3` objects in the same way as it uses its own objects. A user of the `Oviedo3` platform doesn't need to know anything about the COM platform.

INSTANCES

```
b:bool;
wd:word_application;
s:string('hola');
sYES:string('SI');
sNO:string('NO');
c:constream;
```

CODE

```
wd.CheckSpelling(s):b;
jfd b, labelYES;
c.write(sNO);
exit;
```

labelYes:

```
c.write(sYES);
exit;
```

ENDCODE

In this example we can see part of the Carbayon code which uses the “CheckSpelling” method of the `word.application` COM component. The programmer defines the instance “wd” of the class “word\_application” in the same way that defines any other instance. In this case, the class `word_application` is a COM component instead of an `Oviedo3` class. Once the instance has been declared, all the methods of components can be used in the same way as the `Oviedo3` class methods. The programmer doesn't need to know anything about COM except how the naming process is performed in `Oviedo3`. This naming process uses the same name employed by the COM platform but replacing the dot (e.g.: `word.application`) with a underscore (e.g: `word_application`).

## 4 Application to Reuse

From the reuse viewpoint, interoperability between different platforms is a key aspect to promote the development of reusable components [5] able to be used across several platforms or systems. In the past, some proposals such as: megaprogramming [3], COTS [4] or Kits [9], have been proposed to produce components to be used in the construction of software systems as reusable building blocks. In this way, such components don't exhibit the property to be distributed across several platforms. Also, how these components are incorporated in a particular software architecture is not an easy task. From our point of view, the OVIBUS model using the Oviedo3 system increases the degree of reuse because components developed under the Oviedo3 system or any other one supported by OVIBUS, can be developed for reuse purposes in different platforms.

In addition to this, other proposals use adapters [12] to reduce the interaction between components as well increase the reuse level. In our work, the wrapping process can also leverage the adaptation between components in several situations because the component can be reused transparently by a different platform or system.

## 5 Conclusions

In this paper we have presented a model to permit the interoperability among heterogeneous platforms in a transparent mode. Our work improves some of the existing problems associated to the interoperability between platforms or systems. In this way the model presented here improves the interoperability problem because it permits not only the communication between two platforms but also between more than two platforms. Therefore, the scalability of the system is greater. Another conclusion we can obtain from this work is that the communication process between components is performed in a transparent mode. This is possible due to a dynamic wrapping process that improves previous proposals. All of this implies that the reuse level using components belonging from different platforms can be leveraged because the effort to interact such components can be reduced.

Although our results are promising, we have to take into account that the advantage of using a dynamic wrapper produces an overload of the system in the interaction process. Of course, a dynamic wrapping process would be the ideal solution but in real situations we have to consider the system's overload to achieve the right performance.

For a future work we would like to improve the performance of the dynamic wrapping process. Another point is how to add dynamically a new platform to the OVIBUS architecture without recompiling the Oviedo3 system. This is what we call dynamic scalability.

## References

1. Alvarez, D., Tajés, L. Et al: An Object-Oriented Abstract Machine as the Substrate for a Object-Oriented Operating System. 11<sup>th</sup> European Conference on Object Oriented Programming, Jyväskylä, Finland (1997)
2. Bobrow, D.G., Gagriel, R.G., White, J.L.: CLOS in Context – The Shape of the Design Space, In Object Oriented Programming – The CLOS Perspective. MIT Press (1993)
3. Boehm, B., Scherlis, W.L.: Megaprogramming. DARPA Software Technology Conference (1992) 63-82
4. Braun, C.L.: A Lifecycle Process for the Effective Reuse of Commercial Off-theSelf (COTS) Software. 5<sup>th</sup> Symposium on Software Reusability, Los Angeles, CA, USA (1999) 29-36
5. Chan, S.M., Lammers, T.L.: Reusing a Distributed Object Domain Framework. 5<sup>th</sup> International Conference on Software Reuse, Victoria, BC, Canada (1998) 216-223
6. Chappel, D.: Active X and OLE, Microsoft Press, Redmond, Washington, USA (1996)
7. Dominguez, F., Cueva J.M.: Interoperability Oviedo3/COM through Automation. ECOOP'2000 Workshop on Object Interoperability, Sophia, Anthipolis, France (2000) 75-83
8. Edwards, S.H., Weide, B.W., Hollingsworth, J.: A Framework for Detecting Interface Violations in Component-Based Software. 5<sup>th</sup> International Conference on Software Reuse, Victoria, BC, Canada (1998) 46-55
9. Griss, M.L.: PANEL:Architecting Kits for Reuse. 3<sup>rd</sup> International Conference on Software Reuse, Rio de Janeiro, Brasil (1994) 216-217
10. Jones, S.P., Meijer, E., Leijen, D.: Scriptint COM Components in Haskell. 5<sup>th</sup> International Conference on Software Reuse, Victoria, BC, Canada (1998) 224-233
11. Pawlak, R., Melnik, S.: Generic Interoperability Framework, Online at [http://jac.aopsys.com/papers/pawlak\\_tools01.ps](http://jac.aopsys.com/papers/pawlak_tools01.ps)
12. Rine, D., Nada, N., Jaber, K.: Using Adapters to reduce Interaction Complexity in Reusable Component-Based Software Development. 5<sup>th</sup> Symposium on Software Reusability, Los Angeles, CA, USA (1999) 37-43
13. Siegel, J.: CORBA 3: Fundamentals and Programming. John Wiley & Sons (2000)
14. Szyperski, C.: Component Software. Addison-Wesley (1997)