

# Experiments with Reflective Middleware\*

*Fábio M. Costa<sup>†</sup>, Gordon S. Blair and Geoff Coulson*

Distributed Multimedia Research Group  
Department of Computing  
Lancaster University  
Bailrigg, Lancaster  
LA1 4YR, U.K.

e-mail: {fmc,gordon,geoff}@comp.lancs.ac.uk

## Abstract

*Middleware platforms have emerged as an effective answer to the requirements of open distributed processing. Existing standards, however, lack the ingredient of responsiveness in face of the constantly evolving needs of application areas such as groupware, multimedia and mobile computing. In our opinion, to meet the needs of such applications, new approaches to the engineering of middleware platforms are required in order to allow for i) configurability of the underlying support offered by the platform, and ii) the ability to open the implementation in order to inspect and adapt the platform's components. This paper presents an architecture for configurable and open middleware platforms based on the concept of reflection. It also concentrates on the description of a first prototype, which implements the main points of this reflective architecture and demonstrate its practicality.*

---

\*Internal Report MPG-98-11, Computing Department, Lancaster University. Submitted to the Workshop on Reflective Object-Oriented Programming and Systems, ECOOP'98.

<sup>†</sup>Sponsored by CNPq, Brazil

## 1 Introduction

Middleware has emerged as an important architectural component in supporting distributed applications. Its role is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution. The importance of the topic is reflected in the increasing visibility of standardization activities such as the ISO/ITU-T Reference Model for Open Distributed Processing (RM-ODP), OMG's CORBA, the Java RMI, Microsoft's DCOM and the Open Group's DCE.

Although now well established, it is important that such standards remain responsive to new challenges such as groupware, multimedia, real-time, and mobility. To address such challenges, we believe that it is essential that middleware platforms support configurability in order to meet the wide range of different demands posed by the different classes of applications. It is also important that middleware platforms allow for adaptability in the presence of runtime changes in the applications requirements.

To allow for configurability and adaptability, the design of middleware platforms should follow an *open engineering* approach, which exposes the implementation details allowing for the dynamic inspection and manipulation of underlying components. More specifically, we view the concept of *reflection* as a

principled (as opposed to ad hoc) way to provide such openness without compromising the integrity of the platform.

This paper reviews our approach to the design of configurable and open middleware platforms based on the concept of reflection. It presents a language-independent reflective architecture featuring i) a per-object meta-space, ii) the use of meta-models to structure meta-spaces, and iii) a consistent use of object graphs for composite components. It also presents details of a prototype implementation of our reflective architecture; the aim of this prototype is, firstly, to investigate the practicality of the reflective architecture and, secondly, to investigate the (meta) interfaces to be offered to the programmer. The paper is structured as follows. Section 2 provides some background on computational reflection. Section 3 then presents the principles and design of the architecture. Following this, section 4 gives a detailed description of the prototype implementation. Some related work is discussed in section 5, while section 6 offers some concluding remarks and addresses future issues.

## 2 Background on Reflection

The concept of reflection was first introduced by Smith [1]. In this work, he introduced the reflection hypothesis which states:

”In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures”.

The importance of this statement is that a program can access, reason about and alter its own interpretation. Access to the interpreter is provided through a *meta-object protocol (MOP)* which defines the services available at the *meta-level*. Examples of opera-

tions available at the meta-level include altering the semantics of message passing and inserting before or after actions around method invocations. Access to the meta-level is provided through a process of *reification*. Reification effectively makes some aspect of the internal representation explicit and hence accessible from the program. The opposite process is then *absorption* where some aspect of meta-system is altered or overridden.

Smith’s insight has catalysed a large body of research in the application of reflection. Initially, this work was restricted to the field of programming language design [2, 3]. More recently, the work has diversified with reflection being applied in operating systems [4] and, more recently, distributed systems (see section 5).

The primary motivation of a reflective language or system is to provide a principled (as opposed to ad hoc) means of achieving open engineering. For example, reflection can be used to *inspect* the internal behaviour of a language or system. By exposing the underlying implementation, it becomes straightforward to insert additional behaviour to monitor the implementation, e.g. performance monitors, quality of service monitors, or accounting systems. Reflection can also be used to *adapt* the internal behaviour of a language or system. Examples include replacing the implementation of message passing to operate more optimally over a wireless link, introducing an additional level of distribution transparency in a running computation (such as migration transparency), or inserting a filter object to reduce the bandwidth requirements of a communications stream.

## 3 An Architecture for Reflective Middleware

### 3.1 General principles

In common with most of the research on reflective languages and systems, we adopt an *object-oriented model of computation*. As pointed out by Kiczales et al [2], there is an important synergy between reflection and object-oriented computing:

”Reflective techniques make it possible to open up a language’s implementation without revealing unnecessary implementation details or compromising portability; and object-oriented techniques allow the resulting model of the language’s implementation and behaviour to be locally and incrementally adjusted”.

The choice of object-orientation is also important given the predominance of such models in open distributed processing. Crucially, we propose the use of the RM-ODP Computational Model, using CORBA IDL to describe computational interfaces. The main features of this object model are: i) objects can have multiple interfaces, ii) operational, stream and signal interfaces are supported, and iii) explicit bindings can be created between compatible interfaces (the result being the creation of a binding object). The object model also has a sophisticated model of quality of service including QoS annotation on interfaces. Further details of this object model can be found in [5]. In contrast, with RM-ODP, however, we adopt a consistent object model throughout the design (see section 3.3)

The second principle behind our design is to have *per object (or per interface) meta-spaces*. This is necessary in a heterogeneous environment, where objects will have varying capacities for reflection. Such a solution also provides a fine level of control over the support provided by the middleware platform; a corollary of this is that problems of maintaining integrity are minimised due to the limited scope of change (again, see discussion in section 3.3). We do recognise however that there are situations where it is useful to be able to access the meta-spaces of sets of objects in one single action; to support this, we also allow the use of (meta-object) groups (we do not consider this aspect further though in this paper).

As our third principle, we adopt a procedural approach to reflection, i.e. the meta-level (selectively) exposes the actual program that implements the system [6]. This approach has a number of inherent advantages over a more declarative approach. For example, a procedural approach is more primitive (in particular, it is possible to support declarative in-

terfaces on top of procedural reflection but not vice versa). In addition, the required property of causal connection between base-level and meta-level is automatically maintained (as the implementation itself is directly manipulated) [6].

The final principle is to structure meta-space as a number of closely related but distinct meta-space models. This approach was first advocated by the designers of AL-1/D, a reflective programming language for distributed applications [7]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects. The three aspects currently employed are: composition, encapsulation and environment. This is however not a closed list and the range of models may be expanded later. Further details of each of the models can be found below.

## 3.2 Design

### 3.2.1 The structure of meta-space

The overall design of meta-space is illustrated in figure 1. The meta-space is structured as three distinct meta-space models, covering encapsulation, composition and environment. To support this, reflective objects must support a set of operations to reify each of the meta-space models. These operations are encapsulation(), composition() and environment() respectively. Crucially, these operations provide language independent access to the meta-space, although the level of access to each model may be language dependent (we return to this issue below when we look at the details of each individual model).

It is important to realise that, in our approach, objects can have multiple interfaces. Given this, meta-spaces are actually associated with each interface (as opposed to with each object). More precisely, each interface has an associated encapsulation and environment meta-model. The compositional model is however treated differently. In this case, the meta-model is associated with the object itself and is thus common to all interfaces. The compositional meta-model is accessed by calling the composition() operation on any of the interfaces associated with the object. This distinction should become clearer when

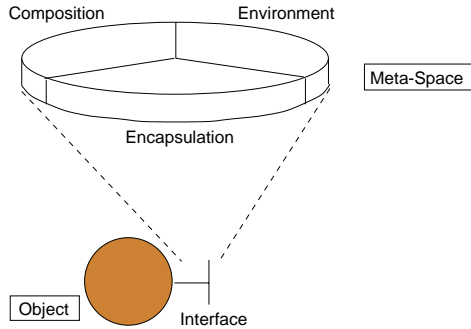


Figure 1: Overall structure of meta-space

the precise details of the different meta-models are considered below.

### 3.2.2 The three meta-models

The compositional meta-model provides access to the object in terms of its constituent objects. The composition of an object is represented as an *object graph*, in which the constituent objects are connected together by *local bindings*. Local bindings are crucial in our design; they provide a language-independent means of implementing the interaction point between interfaces. Importantly, some objects in this graph can be (distributed) binding objects, allowing distributed configurations to be created. In practice, the `composition()` operation returns a graph object with operations to inspect and adapt the composite object, i.e. to view the structure of the graph, to access individual objects in the graph, and to adapt the graph structure and content. For objects that are not composite, the `composition()` operation returns null.

The encapsulation meta-model provides access to the representation of a particular interface in terms of its set of methods and associated attributes, together with key properties of the interface including its inheritance structure. The level of access provided by the encapsulation model is clearly language dependent. For example, with compiled languages such as C access may be limited to inspection of the associated IDL interface. With more open (interpreted) languages, such as Java or Python, more

complete access is possible, such as being able to add or delete methods and attributes. This level of heterogeneity is supported by having a type hierarchy of meta-interfaces ranging from minimal access to full reflective access to interfaces. Note however that it is important that this type hierarchy is open and extensible to accommodate unanticipated levels of access.

Finally, the environment meta-model represents the execution environment for each interface as traditionally provided by the middleware platform. In a distributed environment, this corresponds to functions such as message arrival, enqueueing, selection, dispatching, unmarshalling, thread creation and scheduling (plus the equivalent on the sending side) [3, 8]. Again, different levels of access are supported. For example, a simple meta-model may only deal with the arrival and dispatching of messages at the particular interface. More complex meta-models will allow the insertion of additional levels of transparency or control over thread creation and scheduling. As with the encapsulation meta-model, this level of heterogeneity is accommodated within an open and extensible type hierarchy. Crucially, the environment meta-model is represented as a composite object. Hence, this aspect of the meta-space is inspected and adapted at the meta-meta-level using the graph manipulation objects, i.e. by calling operations on `composition(environment(object))`. Such operations, can be used, for example, to insert a QoS monitor at the required point in the graph.

Note that there is a high level of recursion in the above definition. In particular, the meta-level is realised using object-oriented techniques. Hence, objects/interfaces at the meta-level are also open to reflection and have an associated meta-meta-space. As above, this meta-meta-space is represented by three (meta-meta-) models. Similarly, objects/interfaces at the meta-meta-level have an associated meta-meta-meta-space. This process continues providing an infinite tower of reflection. This is realised in our design by allowing such an infinite structure to exist in theory but only to instantiate a given level on demand, i.e. when it is reified. This provides a finite representation of an infinite structure (a similar approach is taken in ABCL/R [3]). Access to different meta-levels is important in our design although most access

will be restricted to the meta- and meta-meta-levels.

### 3.3 Discussion

In our opinion, the reflective architecture described above provides a strong basis for the design of future middleware platforms and overcomes the inherent limitations of technologies such as CORBA. In particular, the architecture offers principled and comprehensive access to the engineering of a middleware platform. This compares favourably with CORBA which, as stated above, generally follows a black box philosophy with minimal, ad hoc access to internal details.

We also believe that the reflective approach generalises the viewpoints approach to structuring advocated by RM-ODP. As stated above, RM-ODP distinguishes between the Computational Viewpoint (focusing on application-level objects and their interaction) and the Engineering Viewpoint (which considers their implementation in a distributed environment). Crucially, each viewpoint also has its own set of object modelling concepts (for example, the Computational Viewpoint features objects, interfaces and bindings, whereas the Engineering Viewpoint has basic engineering objects, capsules and protocol objects). Consequently, as the models are different, the mapping between the two viewpoints is not always clear. In addition, this approach enforces a two-level structure, i.e. it is not possible to analyse engineering objects in terms of their internal structure or behaviour. Our approach overcomes these limitations by offering a consistent object model throughout, supporting arbitrary levels of openness.

Another benefit of our approach is that it minimises problems of maintaining integrity. This is due to our approach to scoping whereby every object/interface has its own meta-space. Thus changes to a meta-space can only affect a single object. Furthermore, the meta-space is highly structured, again minimising the scope of changes. An additional level of safety is provided by the strongly typed object model. In contrast, the issue of performance remains a matter for further research.

A complete description of our reflective architecture can be found in [9], which presents other impor-

tant aspects not covered here, such as groups, management and the associated component framework.

## 4 Implementation

### 4.1 Overview

We now consider an initial implementation of our reflective architecture. The prototype was implemented in Python 1.5 [10], an object-oriented interpreted language that provides several capabilities for the programmer to open object implementations, such as to dynamically inspect and alter the set of methods that an object supports, or even to change the class of an object at run-time.

The prototype consists of a simple platform based on the concept of *open bindings*, a special kind of composite object that opens the implementation of explicit bindings and allows the programmer to inspect and manipulate the internal behaviour of the binding in a principled way [11].

Note that the MOPs provided for the meta-models are currently rather simple. Nevertheless the design allows for extensibility in future versions. The next subsections present in detail the structure of the basic platform and the approach to the implementation of each of the meta-models.

### 4.2 The basic platform

#### 4.2.1 Conceptual Design

The platform offers services for the explicit establishment of (operational) open bindings between the interfaces of user objects, allowing transparent and customised communication between them. The components of an open binding cooperate to provide the binding behaviour and to present its external interfaces: one interface presents control and management operations to enable explicit interaction with the binding object (such as to start and stop the binding, or to inspect and change its properties), while other interfaces are meant to support interaction (data flow) between user objects. These interfaces are illustrated in figure 2

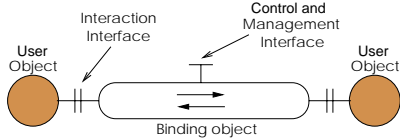


Figure 2: The interfaces of an open binding

An open binding can have several levels of composition, so that one binding may be a component of a higher-level binding. Other typical components are: *stubs*, present in all binding objects and responsible for marshalling/unmarshalling and other kinds of simple data encoding, as well as to provide the binding’s external interfaces; *filters*, (eg. MPEG filters for video encoding/decoding, encryption/decryption filters and compressor/decompressor filters); *monitors*, that observe the flow of data in the binding to check (and maybe enforce) required properties of the binding (eg. quality of service); and *primitive bindings*, directly supported by the underlying infrastructure, provide interfaces to access the network transport protocols (the current version supports primitive bindings based on UDP and TCP).

As an open binding is a distributed object, its components may be located in different capsules, depending on the location of the user objects. The interfaces of co-located components are connected by means of lightweight local bindings (as implemented in [12]). Components are also normally provided with control interfaces (except for primitive bindings in the current implementation), which allow the object to receive and issue control and management messages.

Figure 3 illustrates these concepts showing an external binding, whose components are two stubs and a nested binding object, which is further composed by stubs, filters and a primitive binding.

#### 4.2.2 The Binding Protocol

Another important aspect of the implementation is the binding protocol used by the platform to establish an open binding. This protocol is derived from [5] and is based on the existence of one or more *binding factories*. Each binding factory follows this protocol

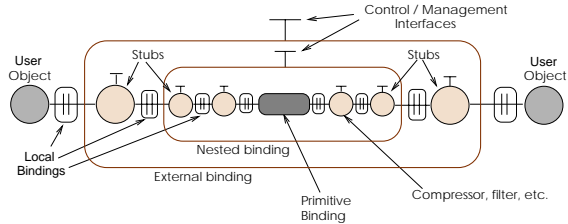


Figure 3: An example open binding

and is specialised to create binding objects with a particular configuration.

The creation of an open binding is started by a requester object, that chooses an appropriate binding factory and asks it to create a new binding. It specifies the name of the interfaces to be bound, as well as optional properties of the binding. Note that the requester can either be one of the participating objects (eg. the client), or a third party object (third party binding).

The binding protocol is then divided into four steps as described below:

- *Step 1 — Locating the interfaces to be bound:* the binding factory calls the name server to resolve the interface names into interface references.
- *Step 2 — Creating the binding:* the binding factory verifies the compatibility between the interfaces to be bound, and then proceeds with the creation of the binding configuration, as follows:
  - If the binding to be created is a *primitive binding*, then it can be instantiated as a protocol stack directly supported by the infrastructure plus a primitive stub at each of its ends.
  - Otherwise, it is a *composite binding* and its creation requires: **(a)** the instantiation of its simple (non-composite) components (stubs, filters, etc) in their respective capsules, and **(b)** invocation of an appropriate binding factory to create the nested binding (following this same protocol).

- *Step 3 — Linking the adjacent interfaces:* the *localBind* operation is called to connect the interfaces of the user objects to the corresponding interfaces of the binding. In addition, all the adjacent components inside the binding (at the same level of composition) are also connected using this same operation.
- *Step 4 — Returning the result:* if all the previous steps have been successful, the binding factory records information about the structure of the new binding object (for future use), and returns an interface reference corresponding to the control interface of the binding; otherwise it returns an exception.

### 4.2.3 Other implementation issues

The management of *interface references* is done by a simple naming service provided by the platform. Every time an interface of an object is created, its name is registered with the name server, which allocates a unique interface reference to it. Interface references have the following Python definition, which comprises information that allow binding factories to locate an interface and to decide about compatibility of interfaces to be bound:

```
class IRef:
    name = ''      # name of this interface
    host = ''     # host IP address
    capsule = ''  # capsule name
    expID = []    # exported operations
    impID = []    # imported operations
    causality = '' # causality of the interface
                  # (client, server, or both)
```

The implementation of multiple interfaces per object is realised through the use of *interface objects*. An interface object contains references to each of the methods exported through the interface; calling the interface results in calling the implementation object itself, without any additional overhead. For the implementation of these simple interface objects, see [12].

Another important issue is the instantiation of objects in remote capsules. Each capsule has a *capsule manager*, which accepts remote calls and pro-

vides services for the creation of objects and for their subsequent handling (eg. to establish a local binding between the interfaces of two adjacent objects). A *remote invocation service*, based on UDP sockets, is provided in order to facilitate remote access to capsule managers and other objects in the infrastructure.

## 4.3 The Compositional Meta-model

### 4.3.1 Introduction

Compositional meta-objects are used to reify the composition aspect of open bindings, and indeed other composite objects. The main purpose of these meta-objects is to maintain the object graphs and provide an interface with facilities for their inspection and manipulation. Importantly, every change in an object graph must be reflected on the corresponding composite object (by the causal connection property).

Object graphs are implemented using the dictionary data structure present in the Python language, which allows for a representation based on the local bindings (the arcs of the graph). For example, the pair *key:value* in the dictionary  $\{(c1, i1) : (c2, i2), \dots\}$  means that there is a local binding between components named *c1* and *c2* which connects their respective interfaces, *i1* and *i2*.

Information about the internal structure of a composite object is obtained from the factory used for its creation (a binding factory in the case of open bindings). This information is used by the compositional meta-object during its initialization to construct the object graph.

As the compositional meta-model associated with an object is common to all of its interfaces, a direct way to implement it is to have a single meta-object, which can be accessed through any of the object's interfaces. (In the case of distributed composite objects, this may require remote access to the meta-object.) A more scalable approach would be to replicate the compositional meta-object and use an object group to maintain consistency among the replicas. For simplicity, however, the former approach is used in the current version of the prototype.

### 4.3.2 The Compositional MOP

Access to the compositional meta-object is given by the platform by means of the operation

**Composition(interface\_name)**

which returns a reference (IRef) to the compositional meta-object associated with the object that implements the given interface. The meta-object is created at the first time the above operation is called for one of the interfaces of the composite object.

Once a client has a reference to the interface of the compositional meta-object, the following operations are available:

**listComponents()** – returns a list of the binding components; for each component, this list gives the identifier of its control interface (which can be resolved to an interface reference), as well as the type of the component (stub, filter, primitive binding, etc).

**getStubs()** – returns the list of stub objects (their names) in the object binding (useful when stopping or starting the binding).

**boundInterface(interface)** – returns the name of the interface that is locally bound to the one given as argument to the call.

**getLBind(obj1,obj2)** – returns the names of the interfaces by which the two objects are locally bound (if there is a local bind between them); this operation is useful when adding a new component.

**addComponent(description, position)** – instantiates a new component (based on its description, that includes the name of its class and its type) and adds it to the composite object (between the two adjacent interfaces specified in the argument **position**). The process of adding a component involves “freezing” the adjacent interfaces, breaking their local binding, and establishing new local bindings between them and the appropriate interfaces in the new component. Note that the addition of a new component only succeeds if the interfaces to be bound match each other.

**delComponent(component\_identifier)** – removes the identified component from the configuration, which involves rebinding the interfaces of the remaining adjacent components.

### 4.3.3 Specific Issues

A problem of integrity may arise when changing the composition of open bindings. As an example, the addition of a single component for data encoding at one side of the binding can make the configuration inconsistent, unless there is a complementary addition of another component (a decoder) at the other side. In the current version, the approach is to rely upon the user of the meta-object for such integrity management (because the criteria that define integrity are normally highly dependent on the application semantics of the binding).

In addition, before initiating composition changes, the user must call the *stop* operation in the control interface of the binding object, in order to prevent any data flow during the process (while the binding is in an inconsistent state). After completing the changes, the *start* operation is called to re-enable the binding.

Regarding this issue, an important extension to the prototype would be the provision of support for the atomic execution of a set of related composition changes. One possible approach would be to replace the above use of *stop* and *start* operations with a mechanism similar to the *multimedia transactions* (transactions that guarantee consistency in re-configurations of multimedia composite objects) proposed in [13]. Another complementary approach would be to augment the compositional MOP with *compound operations* (similarly to those presented in [11]), which would accept the specification of multiple configuration changes and perform them atomically.

## 4.4 The Encapsulation Meta-model

### 4.4.1 Introduction

The implementation of the encapsulation meta-model is based on reflective extensions that we have made to Python [14]. These extensions provide a complete reflective programming model enabling the programmer to dynamically create meta-objects that reify the encapsulation of Python objects. Such meta-objects provide a MOP with operations to inspect methods, attributes and the class of an object, to add and delete methods or attributes, or to change the class

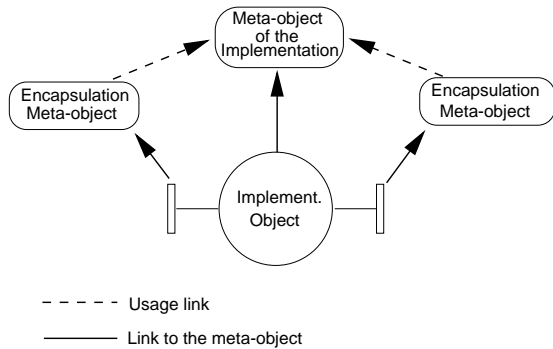


Figure 4: Encapsulation meta-objects and the meta-objects associated with the implementation

of an object. It is also possible to introduce pre- and post-methods to a particular method.

However, Python does not support the concept of interfaces, which means that the above meta-objects are directly associated with the objects, as opposed to the structure of the encapsulation meta-space (see section 3.2.1). Therefore, to implement the meta-model, *encapsulation meta-objects* can be explicitly associated with each interface of an object, reifying only the subset of the object's behaviour that is exposed through the interface. The actions of the encapsulation meta-object are selectively mapped on to the meta-object that is associated with the implementation object<sup>1</sup>, as can be seen in the description of the encapsulation MOP below. The relationship between objects, their interfaces and the two kinds of meta-objects is illustrated in figure 4.

#### 4.4.2 The Encapsulation MOP

Access to the encapsulation meta-objects of an interface is obtained by calling the operation

**Encapsulation(interface\_name)**

provided by the platform. If the meta-object does not exist, it is created as a result of the call.

<sup>1</sup> The creation of this meta-object takes place when the first encapsulation meta-object for one of the object's interfaces is created.

The following operations are available in the encapsulation MOP:

**inspect()** – returns the description of the methods exported by the interface (as specified in the interface's definition).

**addExpMethod(name,function)** – if the method does not exist in the implementation object, its meta-object is called to add the method (using **function** as the method's implementation); then the method is exported through the interface. If the method already exists, only the second step is executed. (At the moment, the only way to really change a method's implementation is to directly access the meta-object of the implementation object and ask it to override its definition; then the method must be removed and re-added to the interface.)

**delExpMethod(name)** – removes the method from the list of exported methods of the interface. The method is not actually deleted from the implementation object (the only way to do that is to directly access the meta-object of the implementation object and ask for the deletion).

**addImpMethod(name)** and

**delImpMethod(name)** – used to add or remove a method description to/from the list of methods that are imported by an interface. It is useful when the base interface is currently bound to another interface whose set of exported methods was changed.

**addPreMethod(name,function)** and

**delPreMethod(name)** – enable the addition and removal of pre-methods, observing the following principle: if a pre-method is added/deleted to a method at an interface, other interfaces that export that same method will not be affected.

**addPostMethod(name,function)** and

**delPostMethod(name)** – allow the addition and removal of post-methods, following the same principle as above.

#### 4.4.3 Specific Issues

Again, special consideration is needed with regard to consistency management. As components in a composite object are related to one another by means of their interfaces, the event of changing the encapsulation of one interface can introduce inconsistency to

the (local or distributed) bindings in which it takes part. Once more, this issue is left to the application to resolve. For example, if a method is added to one interface, the user must call the encapsulation meta-objects of the other bound interfaces, in order to add a correspondent method definition to their lists of imported methods (with `addImpMethod`).

## 4.5 The Environment Meta-model

### 4.5.1 Introduction

The current version of the prototype offers support for a very simple environment meta-object that enables the user to insert and remove *before* and *after* computations to be applied to all invocations on an interface.<sup>2</sup> More specifically, the user can provide Python functions to be executed before (or after) the actual processing of invocations of the interface, where the semantics of this functions can be arbitrary. This essentially provides a hook allowing new behaviour to be introduced to the environment of an interface. For example, a function used as a *before* computation can simply count the number of invocations on an interface, while another can implement a rather complex algorithm for dispatching invocations based on QoS parameters.

The implementation of this functionality is realised by means of an interception mechanism that catches each invocation at an interface and forwards it to the meta-object, where the *before* computation is applied. Then, the meta-object forwards the invocation to the respective method in the implementation object and, after receiving the result, proceeds with the *after* computation. The approach is illustrated in figure 5.

### 4.5.2 The environment MOP

Access to the environment meta-object is obtained by calling the operation

**Environment(interface\_name)**

---

<sup>2</sup>*before* and *after* computations differ from the pre- and post-methods of the encapsulation meta-model in the sense that they apply to the whole interface, irrespective of what method is being called.

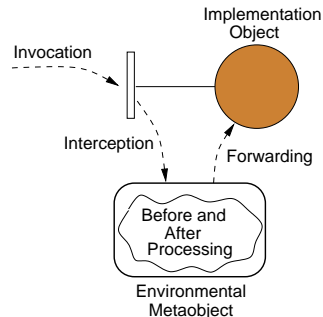


Figure 5: The interception approach to implement an environment meta-object

provided by the platform. If the meta-object does not exist, it is created. During its initialization, the meta-object sets up the interception mechanism for the interface, leaving empty the list of *before* and *after* computations. A reference (IRef) to the interface of the meta-object is returned to the user, who can use it to add functionality to the meta-object. Some basic pre-defined functions for *before* and *after* computation are provided.

The environment MOP in the present version is as follows:

- addBefore(name,function)** and
- addAfter(name,function)** – add a new function to the chain of *before/after* computations for the interface.
- delBefore(name)** and **delAfter(name)** – removes the named function from the chain of *before/after* computations.

### 4.5.3 Specific Issues

Especially worthy to note is the use of environment meta-objects to control the process of sending and receiving data through a reflective open binding. Consider the case of a binding that is meant to support a flow of data from a client to a server object. A meta-object with appropriate *before* and *after* computations can be assigned to the binding interface at the client side, controlling the sending process, whereas the receiving process can be controlled by

another meta-object assigned to the internal interface of the stub object at the server side. This style of intervention is illustrated in figure 6.

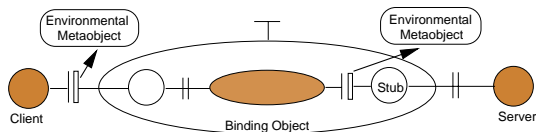


Figure 6: Using environment meta-objects for an open binding

The use of *before* and *after* computation is sufficient for many tasks, such as the one outlined above. However, to support more complex reflective behaviour at the environment meta-object level, we believe that other important elements of the computational model need to be reified, especially: enqueueing of messages, scheduling and dispatching. An implementation of environment meta-objects as composite objects (see section 3.2.2) would allow such extensions to be added dynamically. There is an ongoing effort to incorporate this feature in a new version of the prototype.

## 5 Related Work

There is growing interest in the use of reflection in distributed systems. Pioneering work in this area was carried out by McAffer [8]. With respect to middleware, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) [15]. The level of reflection however is coarse-grained and restricted to invocation, marshalling and dispatching. In addition, the work does not consider key areas such as support for groups or, more generally, bindings. Researchers at APM have developed an experimental middleware platform called FlexiNet [16]. This platform allows the programmer to tailor the underlying communications infrastructure by inserting/ removing layers. Their solution is, however, language-specific, i.e. applications must be written in Java. Manola has carried out work in the design of a "RISC" object model for distributed com-

puting [17], i.e. a minimal object model which can be specialised through reflection. Researchers at the Ecole des Mines de Nante are also investigating the use of reflection in proxy mechanisms for ORBs [18].

Our design has been influenced by a number of specific reflective languages. As stated above, the concept of multi-models was derived from AL/1-D. The underlying models of AL/1-D are however quite different; the language supports six models, namely operation, resource, statistics, migration, distributed environment and system [7]. Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R [3] and CodA [8]. Both these systems feature decompositions of meta-space in terms of the acceptance of messages, placing the message in a queue, their selection, and the subsequent dispatching. Finally, the design of ABCL/R2 includes the concept of groups [19]. However, groups in ABCL/R2 are more prescriptive in that they enforce a particular construction and interpretation on an object. The groups themselves are also primitive, and are not susceptible to reflective access.

Our use of object graphs is inspired by researchers at JAIST in Japan [20]. In their system, adaptation is handled through the use of control scripts written in TCL. Although similar to our proposals, the JAIST work does not provide access to the internal details of communication objects. Furthermore, the work is not integrated into a middleware platform. Similar approaches are advocated by the designers of the VuSystem [21] and Mash [22]. The same criticisms however also apply to these designs. Microsoft's ActiveX software also uses object graphs. This software, however, does not address distribution of object graphs. In addition, the graph is not re-configurable during the presentation of a media stream.

## 6 Concluding Remarks

This paper has presented an approach for the design and implementation of next generation middleware platforms, exploiting the concept of reflection to provide the desired level of configurability and openness in a principled way. We believe that a host of emerg-

ing application areas such as multimedia, real-time systems and mobile computing, will benefit from this approach.

We have also presented the implementation of a prototype, making concrete the compositional, encapsulation and environment meta-models and their realization through meta-objects and the respective meta-object protocols. This first experiment with the reflective architecture has allowed us to identify several issues related to the implementation of the meta-models, such as consistency management and the use of meta-spaces in the context of reflective open bindings.

We are also currently considering some specific areas of future research. The first one consists in the introduction of stream and signal bindings, allowing us to experiment with multimedia applications. In particular the prototype will be integrated with modules for encoding, decoding and the transport of audio and video streams developed by the Adapt project team at Lancaster University [11]. We are currently investigating the implications of this extension.

A second area of research refers to the application of a more comprehensive and standard approach for the composition of (distributed) objects in middleware platforms. One possibility is the use of elements of the JavaBeans component framework (or the emerging CORBA Components model) to guide the definition and use of open bindings and other composite objects.

The longer term goal of this work is to consider a full implementation of a reflective middleware platform, featuring complete language independence. The work will be based on CORBA standard interfaces (in particular, CORBA-IDL will be used for the interface definitions at both base and meta-level).

## Acknowledgements

Fábio M. Costa would like to thank his sponsors, the Brazilian National Council for Scientific and Technological Development (CNPq), and the Federal University of Goiás, Brazil. The research described in this paper is also partly funded by the CNET, France Telecom (CNET Grant 96-1B-239)

and by the EPSRC together with BT Labs (Research Grant GR/K72575). Finally, we would like to acknowledge the contributions of a number of researchers at Lancaster to the ideas described in this paper, namely Anders Andersen (also of the University of Tromsø, Norway), Lynne Blair, Mike Clarke, Philippe Robin, Nikos Parlavantzas, Hector Duran and Katia Saikoski.

## References

- [1] B.C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, MIT, Cambridge, Mass., 1982. Available as MIT Laboratory of Computer Science Technical Report 272.
- [2] G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [3] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA '88*, pages 306–315. ACM, September 1988.
- [4] Y. Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of OOPSLA '92, ACM SIGPLAN Notices*, volume 28, pages 414–434. ACM Press, 1992.
- [5] G.S. Blair and J.B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.
- [6] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA '87*, pages 147–155. ACM, October 1987.
- [7] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/d: A distributed programming system with multi-model reflection framework. In *Proceedings of Workshop on New Models for Software Architecture*, November 1992. (Also available from the Department of Computer Science, Keio University, Japan).
- [8] J. McAffer. Meta-level architecture support for distributed objects. Technical report, Department of Information Science, The University of Tokyo and Object Technology International, 1996. Also published in the proceedings of Reflection'96, G. Kiczales (ed), pp. 39-62, San Francisco.
- [9] G.S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 1998.

- [10] G. van Rossum. *Python Tutorial*, draft release 1.5b1 edition, November 1997. Available at URL <http://www.python.org/1.5/>.
- [11] T. Fitzpatrick, G.S. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. In *Proceedings of International Conference on Configurable Distributed Systems (ICCDs '98)*, Annapolis, Maryland, USA, May 1998.
- [12] A. Andersen. *Local binding through a simple interface*. NORUT Information Technology Ltd., Norway, and Lancaster University, UK, March 1998. Available as <http://starship.skyport.net/crew/anders/dist/tmp/lbind.py.ps>.
- [13] S. Mitchell, H. Naguib, G. Coulouris, and T. Kindberg. Dynamically reconfiguring multimedia components: A model-based approach. In *Proceedings of 8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998. (To be published. Short version available at <http://www.dcs.qmw.ac.uk/research/distrib/Djinn/papers.html>).
- [14] A. Andersen. A note on reflection in python 1.5. Technical report, Department of Computing, Lancaster University, December 1997. Also available at URL: <http://starship.skyport.net/crew/anders/>.
- [15] A. Singhai, A. Sane, and R. Campbel. Reflective orbs: Supporting robust, time-critical distribution. In *Proceedings of ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems*, Jyväskylä, Finland, June 1997.
- [16] R. Hayton. Flexinet open orb framework. Technical Report 2047.01.00, APM, APM Ltd. Poseidon House, Castle Park, Cambridge, UK, October 1997.
- [17] F. Manola. Metaobject protocol concepts for a "risc" object model. Technical Report TR-0244-12-93-165, GTE Laboratories, 40 Sylvan Road, Waltham, MA 02254, USA, December 1993.
- [18] T. Ledoux. Implementing proxy objects in a reflective orb. In *Proceedings of ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation*, Jyväskylä, Finland, June 1997.
- [19] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'91)*, Geneva, Switzerland, 1991. Springer-Verlag. LNCS 512, pp 231-250.
- [20] A. Hokimoto and T. Nakajima. An approach for constructing mobile applications using service proxies. In *Proceedings of 16th International Conference on Distributed Computing Systems*. IEEE, May 1996.
- [21] C.J. Lindblad and D.L. Tennenhouse. The vusystem: A programming system for computer-intensive multimedia. *IEEE Journal of Selected Areas in Communications*, 14(7):1298–1313, 1996.
- [22] S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T-K. Tung, and D. Wu. Towards a common infrastructure for multimedia-networking middleware. In *Proceedings of 7th International Conference on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97)*, St. Louis, Missouri, USA, May 1997.