The Open-ORB Python Prototype API*

Anders Andersen
NORUT IT
aa@computer.org

October 1999

Abstract

This note gives an introduction to the Application Programming Interface (API) of the Open-ORB Python Prototype (OOPP). OOPP is based on the middleware architecture under development in the Open-ORB project at Lancaster University. Open-ORB is a reflective component-based middleware platform. Traditionally, middleware mask out the problems of heterogeneity and distribution. However, a wide range of new and existing applications require the possibilty to configure the support provided and to inspect and adapt this support at run-time. Open-ORB provides this openess through the concept of refelction.

Contents

1 Introduction **1**

2 Programming structures **1**
- 2.1 Interfaces and local bindings 1
- 2.2 Components 3
- 2.3 Binding objects 4

3 Infrastructure **4**
- 3.1 Capsules 4
- 3.2 Name servers 5

4 Meta-models **6**
- 4.1 Encapsulation 6
- 4.2 Composition 8

5 QoS management **8**
- 5.1 Roles 8
- 5.2 Automata 8

*NORUT IT Report IT302/2-99

1 Introduction

The Open-ORB Python Prototype (OOPP) is based on the middleware architecture under development in the Open-ORB project at Lancaster University [1]. This architecture tries to solve the problems of the black-box philosophy of traditional middleware. The requirements from a wide range of applications that need support for (i) multimedia, (ii) real-time, and (iii) mobility can be fulfilled by an open engineering approach in the design of a middleware platform [2]. Open-ORB provides this openness in a principled way (as opposed to ad hoc) through the concept of reflection. A more detailed discussion related to OOPP can be found in [3]. OOPP is available from Starship Python.[1]

Table 1 describes the different arguments and return values used in the descriptions of the interfaces (functions, methods, constructors and so on) in the following text.

2 Programming structures

OOPP provides a set of basic programming structures for the programmer. These structures are influenced by the computational viewpoint of the Open Distributed Processing Reference Model (RM-ODP) [4].

2.1 Interfaces and local bindings

An interface of an object defines a subset of the interactions of that object [5]. A method call to a method of the object and a method call from the object (to a method of another object) are examples of such interactions. Different interface types for operational methods, signals and streams are available. The operational interface type is the basic interface type that all other interface types are based on. It provides a set of exported and imported methods

[1] `http://starship.python.net/crew/anders/oopp/`

i,j	Interface references
[i]	List of interface references
l	Local binding control object
o	Object instance
c,d	Component instances
u,v	Unique component identifier
C	Class or factory
p,q	Capsule proxy (or capsule)
[p]	List of capsule proxies
n	Name server proxy
a,r	Any value/result
k	Key or name
m	Method name
[m]	List of methods
[n]	List of methods
f	Method implementation
t	Argument tuple
w	Argument dictionary
e	Meta-object
L	Any list
D,E	Any dictionaries
x*	x is optional

Table 1: Attributes in interface descriptions

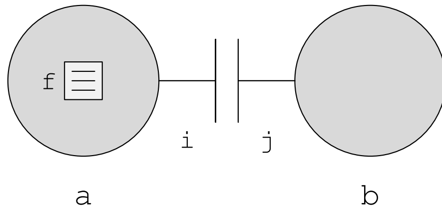

Figure 1: A local binding

and it is associated with a given object. An exported method of an object is a method of that object made available through an interface. An imported method of an object is an external method made available to the object through an interface.

A local binding is an establishing behaviour between two interfaces. A local binding connects the exported method of one interface to the imported method of another interface and vice versa. The two objects a and b in Figure 1 have an interfaces i and an interface j, respectively. Interface i and j are bound with a local binding. Object b can call method f of object a through its interface j.

The following services for interfaces and local bindings are provided by the lbind module:

```
IRef(o,[m],[n])→i
localBind(i,j)→l
localBindOneWay(i,j)→l
breakBinding(i,j)
l.breakBinding()
l.reBind(i,j)
l.reBindOneWay(i,j)
```

IRef is the class used to create interfaces and localBind is a function that creates a local binding between two interfaces. The interfaces and the local binding between object a and b in Figure 1 are created with the following Python code:

```
i = IRef(a,["f"],[])
j = IRef(b,[],["f"])
l = localBind(i,j)
r = j.f(2)
```

The arguments to the IRef constructor is (i) the object, (ii) the description of exported methods and (iii) the description of imported methods.[2] The localBind function returns a local binding control object that can be used to control the local binding. A local binding only exists as cross-references in the interfaces bound. A local binding can be broken with the breakBinding function or with the breakBinding method of the local binding control object. The following code illustrates these two methods for the example above (select one):

```
breakBinding(i,j)
l.breakBinding()
```

It is possible to create a one-way local binding with the localBindOneWay function. The semantics are that a connection is made from the imported methods of one interface to the exported methods of the other interface and *not* vice versa. A local binding control object representing a broken binding can be used to create a new local binding with the reBind and reBindOneWay methods.

Specialised stream and signal interfaces are also available (see the sigbind and streambind module below). These interfaces have a source and a sink pair. A local binding between a source and a sink can be created with the localBind function. The code below creates a stream interface reference source and sink pair and makes a local binding between them. The source interface is associated with object a and the sink interface is associated with object b:

```
src = StreamSrcIRef(a)
sink = StreamSinkIRef(b)
l = localBind(src,sink)
src.put(data)
```

[2] In this prototype the descriptions of exported and imported methods are lists of method names. This can be extended with an interface description language like the CORBA IDL.

A stream interface uses a put method with one data argument. Object b in the example above implements the put method. A signal interface pair is created with the SigSrcIRef and the SigSinkIRef classes. The signal source interface provides (exports) an event method without any arguments.

2.2 Components

A component is a unit of independent deployment [6]. They are developed and delivered independently and provide access to their requested and provided services (methods) through one or more specified interfaces. All interactions between components are specified through these well defined interfaces. Such an interface connection architecture has a basic conformance criteria that says that the system's components interacts only as specified in their interfaces [7].

The interfaces of a component are available without any previous knowledge about the component providing them. The interfaces provided can be browsed and a specific interface can be accessed by its key (name). A primitive component that encapsulates one object and provides/requests access to/from its object through a set of interfaces can be created with the Component class or the componentFactory factory from the component module:

```
Component(D,o)→c
componentFactory(L,C,t*,w*)→c
```

A component for object a in the example above can be created with the following statement:

```
ca = Component({"op":i},a)     1
```

The result is a component ca with an interface with key "op" represented by the interface reference i that exports the method f. The component encapsulates object a that implements method f. The interfaces of a component are available in the interfaces attribute of the component. Interface "op" of ca can be accessed with the ca.interfaces["op"] expression.

A class A has a constructor with no arguments that creates an object with an interface in attribute i. A component with an interface with key "i" can the be created with the following statement:

```
ca = componentFactory(["i"],A) 1
```

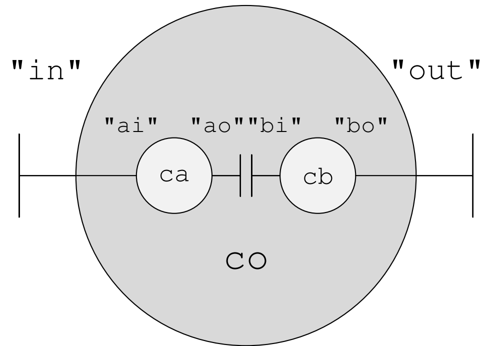

Figure 2: A composite component

A composite component is a way to manage a complex component. In particular, a composite component encapsulates a graph of components. The outside view of a composite component is similar to a primitive component: it provides a set of interfaces that can be browsed and accessed through their keys (names). Figure 2 illustrates a simple composite component co providing the external interfaces "in" and "out". It contains two components ca and cb connected with a local binding between their interfaces "ao" and "bi". The external interface "in" is a mapping to interface "ai" in component ca and the external interface "out" is a mapping to interface "bo" in component cb.

Composite components can be created with the generic Composite class or the generic compositeFactory factory from the composite module:

```
Composite(D,E,a*)→c
compositeFactory(D,E,a*)→c
```

The first argument of the Composite class constructor is the external interfaces of the component and the second specifies the component graph. The component graph is specified by the components contained in the composite component ("comps"), the set of internal interfaces ("iif") and the set of edges (local bindings) between these internal interfaces ("edges"). The composite component co above was created with the following statement:

```
co = Composite(                    1
   {"in":(ca,"ai"),                2
    "out":(cb,"bo")},              3
   {"comps":[ca,cb],               4
    "iif":{"ao":(ca,"ao"),         5
           "bi":(cb,"bi")},        6
    "edges":[("ao","bi")]})        7
```

The Composite class and compositeFactory factory can be used to create any composite component. The result is a fairly complex constructor syntax. A composite component class or factory is usually a refinement of the generic composite class and factory with an less complex constructor syntax. The binding objects discussed below are created with such refined classes or factories.

2.3 Binding objects

Local bindings can only be used between interfaces in the same address space (see capsule below). In contrast, binding objects can be used to create bindings between interfaces in different address spaces or even on different nodes. An operational binding is meant to replace a local binding in such cases. An operational binding is a specialised composite component called a binding object [8]. Binding objects for streams and signals are also available in OOPP. An operational binding can be created with the constructor of the OpBinding class or created and installed with the remoteBind function (both from the opbind module):

```
OpBinding(i,j,a*)→c
remoteBind(i,j,a*)→c
```

The difference between this class and function is that the OpBinding constructor only creates the binding object while the remoteBind function also binds the external interfaces of the binding object to the two interfaces to be bound (in the same way as localBind binds two bindings to be bound). Object a with interface i in capsule A exports its method f. Object b in capsule B with interface j imports method f. The following code creates an operational binding between interface i and j and then calls method f of object a in capsule A through interface j in capsule B:

```
b = remoteBind(i,j)        1
r = j.f(2)                 2
```

A signal binding can be created with the constructor of the SigBinding class, but as argued above for remoteBind they can more easily be created and installed with the sigBind function. The sigbind module provides the following classes and functions:

```
SigSrcIRef(o)→i
SigSinkIRef(o)→i
SigBinding(p,q,a*)→c
sigBind(i,j,a*)→c
```

The sigBind function creates a (remote) signal binding between a signal source interface and a signal sink interface.

The stream binding provided by OOPP does not provide any retransmission, sequencing or buffering. Every data frame fed in to it from the source side will be delivered as quickly as possible to the sink side. However, this approach is too simple for many applications that need a more advanced stream binding. The stream binding provided, however, can be used as a starting point for the creation of a binding with the features needed for the given application. Stream bindings can be created with the constructor of the StreamBinding class, but as argued above for remoteBind they can more easily be created and installed with the streamBind function. The streambind module provides the following classes and functions:

```
StreamSrcIRef(o)→i
StreamSinkIRef(o)→i
StreamBinding(p,q,a*)→c
streamBind(i,j,a*)→c
```

The streamBind function creates a stream binding between a stream source interface and a stream sink interface. The following code creates a stream binding between the stream source interface src and the stream sink interface sink and then sends 100 data frames through the binding to the sink:

```
b = streamBind(src,sink)        1
for i in range(100):            2
    src.put(data[i])            3
```

3 Infrastructure

The infrastructure is a supporting environment for programs in OOPP. The infrastructure is influenced by the engineering viewpoint of RM-ODP [8], but the engineering viewpoint of RM-ODP is also related to the meta-models in OOPP (see section 4).

3.1 Capsules

Each address space in OOPP is controlled by a capsule. A capsule provides services for its local components (the components located in the address space it controls). It can also provide services to remote components through a capsule proxy. A capsule proxy and a local capsule have identical interfaces, but the requests through a capsule proxy are forwarded to the capsule it represents and the

replies are returned back to the caller through the proxy. The local capsule is available through the `local` attribute of the `capsule` module and a capsule proxy for a remote capsule is created with the `CapsuleProxy` class:

```
CapsuleProxy(a,r)→p
```

The following services are provided by a capsule (and a capsule proxy) `p`:

```
p.serve()
p.servethread()
p.stopserve()
p.registerComponent(c)→u
p.mkComponent(C,t*,w*)→u
p.rcpComponent(u,p)→v
p.mvComponent(u,p)→v
p.delComponent(u)
p.getIRef(u,k)→i
p.localBind(i,j)→l
p.localBindOneWay(i,j)→l
p.breakBinding(i,j)
p.callMethod(u,k,m,t*,w*)→r*
p.announceMethod(u,k,m,t*,w*)
p.announceThread(u,k,m,t*,w*)
p.sendMethod(u,k,m,t*,w*)→a
p.recvMethod(a)→r*
p.newport(a)→r
p.delport(r)
```

A component has to be registered in the capsule to be available for the services provided. A component created with the `mkComponent` method of the capsule will automatically be registered in the capsule. The `registerComponent` and `mkComponent` methods return a local unique identifier for the component (unique in the capsule). This identifier together with a capsule proxy provides a global identification of a component. The code below uses a capsule proxy `p` to create an instance of the component class `C` in a remote capsule (the constructor of class `C` is called without any arguments). The result is a component with the unique identifier `u`. It then creates an operational binding between the local interface `j` and the remote interface `i` of component `u`. It is then possible to call method `f` of the remote component `u` through the interface reference `j`.

```
u = p.mkComponent(C)        1
i = p.getIRef(u,"i")        2
b = remoteBind(i,j)         3
r = j.f(2)                  4
```

The `getIRef` service of a capsule returns a global interface reference. In the example above is the interface reference of interface `"i"` in component `u` returned. The `getIRef` service of a capsule should always be used when a global interface reference of a registered component is needed.[3]

The capsule also provides services to establish and break a local binding in the capsule. These services are useful when a local binding in a remote capsule has to be established or broken. The request is then done through a capsule proxy.

The `callMethod` service of an capsule is a low-level method used to call a specific method in an interface of a registered component. This service is meant for implementing higher level bindings or services. The application programmer should use a binding object to establish a connection between two interfaces and call the remote method through the interfaces and the binding.

The following code illustrates the difference between using `callMethod` (line 1) and a operational binding (line 4). Method `f` is in both cases called with an argument `2` an the result is saved in `r`:

```
r = callMethod(u,"i","f",(2,))  1
i = p.getIRef(u,"i")            2
b = remoteBind(i,j)             3
r = j.f(2)                      4
```

The capsule also provides other low-level methods not listed above. This includes announcements method calls, asynchronous method calls and services to duplicate and move components. A capsule proxy can only be used to access a capsule if this capsule has started its serving loop. This has to be done explicitly with the `serve` method of the local capsule.

3.2 Name servers

A name service is needed to make it possible for components in different capsules to interact. The simple name service provided in OOPP is implemented with a name server. Interfaces and capsules can be registered by a key (name) in such a name server. The name server is accessed through a name server proxy. A name server proxy for a given name server can be created with the knowledge of the location (the node) of the name server and the port the

[3] A global interface reference is useful outside the local capsule of the interface. It contains a 'unique component identifier' 'capsule proxy' pair to identify the component it is associated with (and its location).

name server is listening on (a default port is used when it is not explicit given). The name server provides the following services:

```
n.exportIRef(k,i)
n.exportCaps(k,p)
n.delIRef(k)
n.delCaps(k)
n.lookupIRef(k)→i
n.importIRef(k)→i
n.importCaps(k)→p
n.listIRefs()→[i]
n.listCaps()→[p]
```

The `exportIRef` method makes the interface references available through a key for everyone that have access to this name server (through a name server proxy). The `lookupIRef` method returns the interface reference exported with the given key. The `importIRef` creates an implicit operational binding to the exported interface and returns an interface reference to an interface connected to the opposite side of this implicit binding. The returned interface reference can be used immediately to call methods exported in the exported interface reference (and vice versa). Almost the equivalent set of services are available for capsules. The `importCaps` method returns a capsule proxy. A lookup method for capsules does not exists because capsules are only accessed through implicit bindings.

4 Meta-models

OOPP implements two of the Open-ORB meta-models: the encapsulation and the composition meta-model. The implementation of these meta-models i OOPP gains a lot from the reflective features of Python [9].

4.1 Encapsulation

The encapsulation meta-model provides access to the representation or the implementation of interfaces and objects (components). The encapsulation meta-model of a given object or interface is accessed through its encapsulation meta-object. A meta-object does not exists until it is accessed. The `encapsulation` function is used to get access to the encapsulation meta-object of an object or an interface. This is the most common services of the encapsulation meta-object:

```
e.inspect()→D
e.getattr(k)→r
e.setattr(k,a)
e.delattr(k)
e.addMethod(k,f)
e.delMethod(k)
e.addXxxMethod(k,f)
e.delXxxMethods(k,f)
e.addGetAttr(k,f)†
e.delGetAttr(k,f)†
e.addSetAttr(k,f)†
e.delSetAttr(k,f)†
e.changeClass(C)†
e.addImpMethod(k)‡
e.delImpMethod(k)‡
e.addImpXxxMethod(k,f)‡
e.delImpPreMethod(k,f)‡
e.changeObject(o)‡
e.restore()
```

The services marked with † are only available from the meta-objects of objects and services marked with ‡ are only available from the meta-objects of interfaces. The services listed without any marks have effect on the exported methods when they are applied on interfaces.

The `inspect` method returns a detailed description of the object or the interface. The `restore` method removes the encapsulation meta-object.

New methods can be added to an object or to the exported methods of an interface with the `addMethod` service. A new method `get` that returns the value of the attribute `x` is added to object `o` with this code:

```
def getx(self):                 1
    return self.x               2
eo = encapsulation(o)           3
eo.addMethod("get",getx)        4
print o.get()                   5
```

The `addXxxMethod` listed above represents the three methods `addPreMethod`, `addPostMethod` and `addWrapMethod` that are used to add pre-, post- and wrap-methods, respectively. A pre-method is a method that is called before the actual method is called. The pre-method has access to the arguments of the method and it can read and change their values. A post-method is a method that is performed after the actual method has returned. It has access to the arguments and the return value of the method and it can change the return value before it is passed to the caller. A wrap-method is wrapping the actual method. It can manipulate the arguments and the return values. It can even decide not to call the actual method at all.

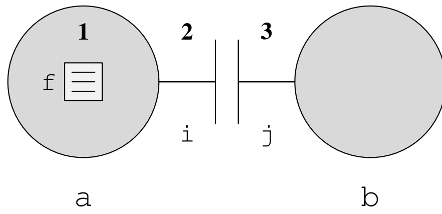

Figure 3: Method adaption

Below is a post-method added to the `get` method of object `o`. The post-method prints the return value of `get` before it returns to the caller:

```
def p(self,m):                    1
     print m["result"]            2
eo = encapsulation(o)             3
eo.addPostMethod("get",p)         4
```

The function `p` in the code above has two arguments. All pre-, post- and wrap-methods have to have these two arguments. The first argument `self` is a reference to the instance of the object (as in every method of an object in Python). The second argument `m` is a dictionary containing information about the method call. This includes a reference to the object, the name of the method, the method, the arguments and the return value (`"result"`).

The `addImpMethod` is used to add a method to the list of imported methods of an interface. The usage of `addImpMethod` can be demonstrated together with the `addMethod` service for interfaces. (the semantics of the `addMethod` service for interfaces are "add an exported method to the interface"). Object `o` has an interface `o.i` that exports some of the methods of its object and is bound with a local binding to another interface `j`. The code below adds method `get` from object `o` to these interfaces. Notice that `addImpMethod` also updates the local binding.

```
ei = encapsulation(o.i)           1
ei.addMethod("get",o.get)         2
ej = encapsulation(j)             3
ej.addImpMethod("get")            4
print j.get()                     5
```

Figure 3 illustrates the usage of all the different services used to add pre-, post-, and wrap-methods. The example contains an object `a` with a method `f`. Interface `i` exports method `f` and interface `j` imports method `f`. Interface `i` and `j` are connected with a local binding. The encapsulation meta-object `ea` for `a` can be used to add pre-, post- and wrap-methods to `f` in object `a` (**1** in Figure 3). The encapsulation meta-object `ei` for `i` can be used to add pre-, post- and wrap-methods to the exported method `f` in interface `i` (**2** in Figure 3). Finally, the encapsulation meta-object `ej` for `j` can be used to add pre-methods, post-methods and wrap-methods to the imported method `f` in interface `j` (**3** in Figure 3). The example below adds a post-method at each point **1** (line 4), **2** (line 5) and **3** (line 6) in Figure 3. All the added post-methods print out the return value ("result") before they increase it with one.

```
def inc(self,m):                      1
     print m["result"],               2
     m["result"]=m["result"]+1        3
ea.addPostMethod("f",inc)             4
ei.addPostMethod("f",inc)             5
ej.addImpPostMethod("f",inc)          6
print j.f(2)                          7
```

Suppose method `f` with argument 2 in object `a` returns the value 1. The output from the code above will then be "`1 2 3 4`", where 4 is the final return value.

Objects have attributes and the encapsulation meta-object can install methods to be called when the attributes of an object are read or changed. These features are useful for different monitoring tasks. One possibility is to monitor when a given attribute becomes greater than a limit value and then raise an alarm. The example below adds a method to print a warning message when attribute `x` of object `o` becomes greater than 10. The statement in line 6 will generate a warning message.

```
def warning(self,k,v):                1
     if v > 10:                       2
          print "Warning"             3
eo = encapsulation(o)                 4
eo.addSetAttr("x",warning)            5
o.x = 11                              6
```

The first argument of the `addSetAttr` method is the name of the attribute in the object. The second argument is the function to call when attribute `x` of object `o` is given a new value. This function has three arguments, where the first is the object instance (equal to `o` in this case), the second is the name of the attribute (`"x"` in this case) and the last argument is the new value of the attribute (11 in this case).

The `changeClass` method makes it possible to change the class of the object at any time. The

`changeObject` method actually change the object associated with the interface. These powerful features must be used carefully.

4.2 Composition

Complex binding objects are typical composite components. Multimedia applications in mobile environments are examples where the component graph of composite components need to be manipulated and restructured (changed and extended) during their life cycle. The composition meta-model is provided for this kind of manipulation of composite components.

The composition meta-model of a composite component is accessed through its composition meta-object. The `composition` function is used to get access to the composition meta-object of a composite component. The services provided are influenced by the operations originally proposed in the Adapt project for the manipulation of object graphs of open bindings [10]. The following operations are available from the composition meta-object:

```
e.inspect()→D
e.add(c)
e.remove(c)
e.bind(i,j)
e.break(i,j)
e.replace(c,d)
```

The `inspect` method of a composition meta-object returns a description of the composite component including the contained objects and the edges of the component graph. The `add` method adds a new component to the composite component, but no new edges (local bindings) are created. The new component can be located in a remote capsule. The `bind` method creates a new edge in the component graph and the `break` method breaks such a binding. It is transparent for the user if the actual local binding is created or removed locally or in a remote capsule. The `replace` method replaces an existing component with a new one. The new component must have a matching set of interfaces.

5 QoS management

OOPP includes support for quality of service (QoS) management. The management is done with a set of management objects connected with signal bindings. A management object can use the meta-space of the components under management to change the behaviour of these components. This is done to fulfil the goal of the given management policy.

5.1 Roles

A management object can have three different roles. A *monitor* collects and filters information from the running system. For example, a monitor of a buffer could check if buffer overflow occurs to often (with a given definition of 'to often'). A *strategy selector* collects information from the monitors and, based on this information and on timing constraints, decide to select a management strategy. For example, a strategy selector that receives a buffer "overflow to often" signal from a monitor could decide to delay the stream at its source. The *strategy activator* activates (performs) the selected strategy when it receives a signal from a strategy selector. The monitors and the strategy activators can use the meta-spaces of the components under management to perform their tasks.

Figure 4 illustrates a management setup for a producer/consumer case. The monitor `M` monitors if there is a buffer "overflow to often". The strategy selector `S` selects strategy `P` or `C`. Strategy activator `C` manipulates the meta-models of the consumer and strategy activator `P` manipulates the meta-models of the producer.

5.2 Automata

The monitors and strategy selectors are implemented with automata components. Their behaviour is specified in formal timed automata descriptions. One of the advantages of using automata is that we can reason about and simulate their behaviour. This, together with a formal description of the complete system can be used to simulate and formally reason about the whole system [11, 12].

A more detailed quality of service management example can be found in [13]. This also includes a detailed evaluation of the approach.

Acknowledgements

Frank Eliassen has been my supervisor during this project. His contribution has been important for the final outcome of the project. Gordon Blair has been the main supplier of ideas and feedbacks during the design and implementation of OOPP. Michael Papathomas has contributed a lot to the early attempts to introduce reflection to the programming model. The contributions from Geoff Coulson in the Open-ORB

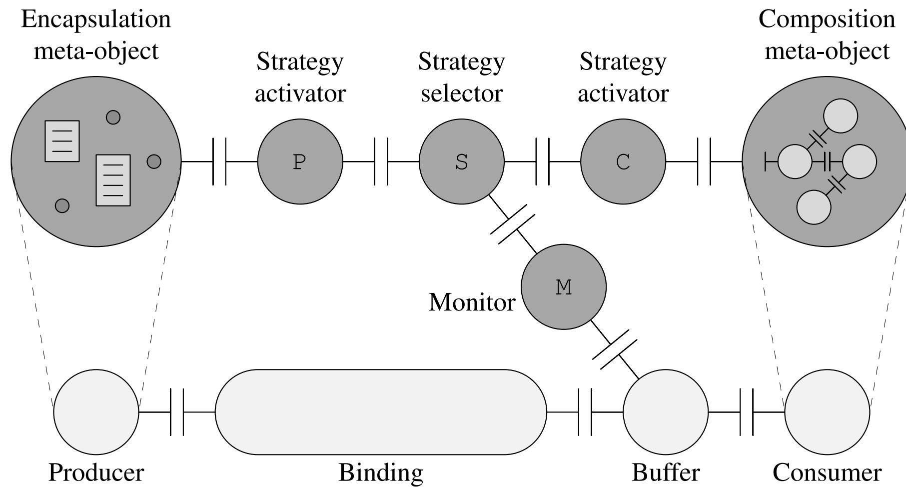

Figure 4: A management setup for the producer/consumer case

project had a great influence in this work. Fabio Costa has also been an important discussion partner in the realisation of the current programming model and meta-models. David Sánchez had the pleasure of being the important first user of OOPP. His feedback has been valuable for the actual implementation. Lynne Blair has contributed a lot to the automata-based quality of service management implemented in OOPP. Finally, thanks to the various members of the Distributed Multimedia Research Group at Lancaster University that contributed to discussions on areas related to this work.

References

[1] Gordon S. Blair, Geoff Coulson, Philippe Robin, and Michael Papathomas. An architecture for next generation middleware. In *Middleware'98*, September 1998.

[2] Frank Eliassen, Anders Andersen, Gordon S. Blair, Fabio Costa, Geoff Coulson, Vera Goebel, Øyvind Hanssen, Tom Kristensen, Thomas Plageman, Hans Ole Rafaelsen, Katia B. Saikoski, and Weihai Yu. Next generation middleware: Requirements, architecture, and prototypes. In *7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '99)*, Tunisia, South Africa, December 1999.

[3] Anders Andersen, Gordon S. Blair, Geoff Coulson, and Frank Eliassen. A reflective component-based middleware in Python. Technical report, NORUT IT, 1999. Submitted IPC8.

[4] ISO/IEC. Open distributed processing reference model, part 1: Overview. ITU-T Rec. X.901 — ISO/IEC 10746-1, ISO/IEC, 1995.

[5] ISO/IEC. Open distributed processing reference model, part 2: Foundations. ITU-T Rec. X.902 — ISO/IEC 10746-2, ISO/IEC, 1995.

[6] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley, 1998.

[7] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. Technical Report CSL-TR-95-674, Computer Systems Laboratory, Stanford University, July 1995.

[8] ISO/IEC. Open distributed processing reference model, part 3: Architecture. ITU-T Rec. X.903 — ISO/IEC 10746-3, ISO/IEC, 1995.

[9] Anders Andersen. A note on reflection in Python 1.5. Distributed Multimedia Research Group Report MPG-98-05, Lancaster University, UK, March 1998.

[10] Tom Fitzpatrick, Gordon S. Blair, Geoff Coulson, Nigel Davies, and Philippe Robin. Supporting adaptive multimedia applications through open bindings. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS '98)*, Annapolis, Maryland, US, May 1998.

[11] Lynne Blair. *The Formal Specification and Verification of Distributed Multimedia Systems*. Dr. thesis, Department of Computer Science, Lancaster University, September 1994.

[12] Gordon S. Blair, Lynne Blair, Howard Bowman, and Amanda Chetwynd. *Formal Specification of Distributed Multimedia Systems*. UCL Press, 1998.

[13] David Sánches Gancedo. QoSMonAuTA, QoS monitoring and adaptation using timed automata. Master's thesis, Lancaster University, UK, September 1999.