

OOPP: A reflective component-based middleware

Anders Andersen*
Department of Computer Science
University of Tromsø
N-9037 Troms, Norway
aa@computer.org

Gordon S. Blair†
Computing Department
Lancaster University
Lancaster LA1 4YR, UK
gordon@comp.lancs.ac.uk

Frank Eliassen†
Department of Informatics
University of Oslo
N-0316 Oslo, Norway
frank@ifi.uio.no

Abstract

Traditional middleware, like CORBA and Java RMI, are not flexible and adaptable enough for new application types that need support for multimedia, real-time and mobility. The next generation middleware platforms should be designed from the beginning with flexibility and adaptability in mind. This can be done by adopting an open engineering approach for the design of the middleware platform. Reflection provides a principled (as opposed to ad hoc) means of achieving open engineering. The Open-ORB Python Prototype (OOPP) presented in this paper implements a flexible and adaptable middleware based on the principle of reflection.

1 Introduction

The role of middleware is to present a unified programming model to application writers and mask out the problems of heterogeneity and distribution [1]. However, the middleware has to remain responsive to challenges and demands from existing and new type of applications, including (i) support for multimedia, (ii) real-time requirements, and (iii) increasingly mobility [2, 3]. Given this, it should be possible to configure the underlying support offered by the middleware platform to satisfy the requirements from a wide variety of applications. Example of such configurations are scheduling policies, special protocols for multimedia, and resource management.

Another important requirement is the possibility to inspect and adapt the support offered at run-time. This can be done by adopting an open engineering approach for the design of the middleware platform. Reflection provides a principled (as opposed to ad hoc) means of achieving such open engineering.

Reflection initially emerged in the programming language community as an important technique to introduce more flexibility and openness into the design of programming languages. This work tries to use these techniques to introduce flexibility and openness into the area of distributed systems, in general, and middleware, in particular.

The aim of this paper is to present a Python prototype implementation of such a middleware platform. The Open-ORB Python Prototype (OOPP) is based on an architecture under development in the Open-ORB project [4]. The paper is structured as follows. Section 2 gives an introduction to the Open-ORB architecture. Section 3 presents the programming structures and section 4 presents the infrastructure of the prototype. Section 5 shows how reflection is provided through the implementation of two meta-models. Section 6 discusses how quality of service management is achieved in the implemented prototype. Section 7 concludes the paper.

* Also at NORUT IT, N-9291 Tromsø, Norway, during this work.

† Adjunct Professor at University of Tromsø.

2 Open-ORB

2.1 Reflective middleware

The reflection hypothesis introduced by Smith in 1982 [5] 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 operation 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. Initially, Smith's work catalysed a large body of work to the field of programming language design [6, 7, 8]. Later reflection has been applied to operating systems [9] and, more recently, distributed systems [10].

Current middleware platforms adopt a black box philosophy, whereby the implementation details are hidden from the application programmer. However, there is increasing evidence though that the black box philosophy is becoming untenable. For example, the OMG have recently added internal interfaces to CORBA to support services such as transactions and security, and the Portable Object Adapter is another CORBA attempt to introduce openness. All these approaches are rather ad hoc. The Open-ORB project tries to address this by providing openness in the middleware design in a principled way through the concept of reflection.

2.2 General principles

Open-ORB adopts a component-based model of computation. A component is a unit for composition and independent deployment [11]. This is important when possible replacements and restructuring of components in a system are determined. In Open-ORB components are used for the structuring of the middleware platform itself; deployable components are the foundation for flexibility (reconfiguration) of Open-ORB. The Open-ORB component model is derived from the computational viewpoint of RM-ODP [12] where (i) components have interfaces, (ii) interfaces for continuous media are supported and (iii) explicit bindings can be created between compatible interfaces.

Open-ORB adopts a procedural (as opposed to declarative) approach to reflection. The meta-level exposes the actual implementation (and not a set of declarative statements describing it). This primitive approach is more flexible than a declarative approach and it makes it possible to access the meta-level of a meta-level (the meta-meta-level). This can be used to inspect and adapt the meta-level. In fact, an infinite structure of meta-levels can in principle exist since meta-levels are instantiated on demand.

In addition, Open-ORB supports per interface (or sometimes per component) meta-spaces. This is necessary in a heterogeneous environment where components 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. In situations where it is useful to access sets of meta-spaces in a single action the use of groups are possible [13].

Finally, Open-ORB structures the 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 [14]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects.

2.3 Meta space

In reflective systems, structural reflection is concerned with the content of a given component [7]. In Open-ORB, this aspect of meta-space is represented by two distinct meta-models, namely the encapsulation and composition meta-models. The encapsulation meta-model provides access to the representation of a particular interface or component in terms of its set of methods, attributes and other key properties (including its inheritance structure). This is equivalent to the introspection facilities available, for example, in the Java language, although we go further by also supporting adaptation. The composition meta-model provides access to the component graph of a component. The component graph can be inspected and altered using this model. All interfaces of a component provides the same composition meta-model.

Behavioural reflection is concerned with activity in the underlying system [7]. This is represented the environmental meta-model. In terms of middleware, the underlying activity equates to functions such as message arrival, enqueueing, selection, unmarshalling and dispatching (plus the equivalent on the sending side) [7, 10].

Multimedia, real-time and mobile applications need access to the resources and the resource management of the system. Open-ORB introduces the resource meta-model [15] for this task. This model is concerned with the allocation and management of resources associated with any activity in the system.

2.4 OOPP

A prototype implementation of the suggested Open-ORB middleware architecture has been implemented in Python.¹ The Open-ORB Python Prototype (OOPP) [16] is influenced by the Open Distributed Processing Reference Model (RM-ODP) [17, 18]. In addition, it provides structural reflection through the encapsulation and composition meta-models.

3 Programming structures

3.1 Interfaces and local bindings

An interface of an object defines a subset of the interactions of that object [12]. 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 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`.

`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 Python code listed in the same Figure. 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

¹<http://www.python.org/>

²In 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.

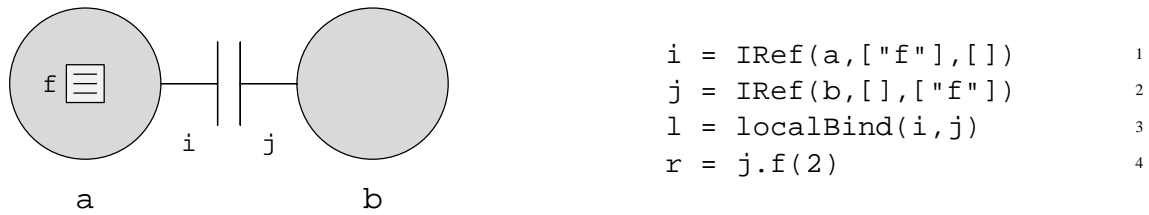


Figure 1: A local binding

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 interface reference contains a description of the interface including its exported and imported methods.

Specialised stream and signal interfaces are also available. 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. A stream interface pair is created with the `StreamSrcIRef` and the `StreamSinkIRef` classes and signal interface pair is created with the `SigSrcIRef` and the `SigSinkIRef` classes.

3.2 Components

A component is a unit of independent deployment [11]. 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 interact only as specified in their interfaces [19].

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. A component for object `a` in the example above can be created with the statement `ca = Component({ "op" : i }, a)`. The result is a component `ca` with an interface with the key `"op"` and the interface reference `i`. The component encapsulates object `a` that implements method `f` exported in interface `i`. The interfaces of a component are available in the `interfaces` attribute of the component. Interface `"op"` of `ca` can be accessed with the expression `ca.interfaces["op"]`.

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. The composite component `co` above was created with the Python code listed in Figure 2. The first argument (line 2–3) is the external interfaces of the component and the second (line 4–7) 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` class can be used to create any composite component. The consequence is a complex constructor syntax. A composite component class is usually a refinement of the `Composite` class with an less complex constructor syntax. The binding objects discussed below are examples of such composite components.

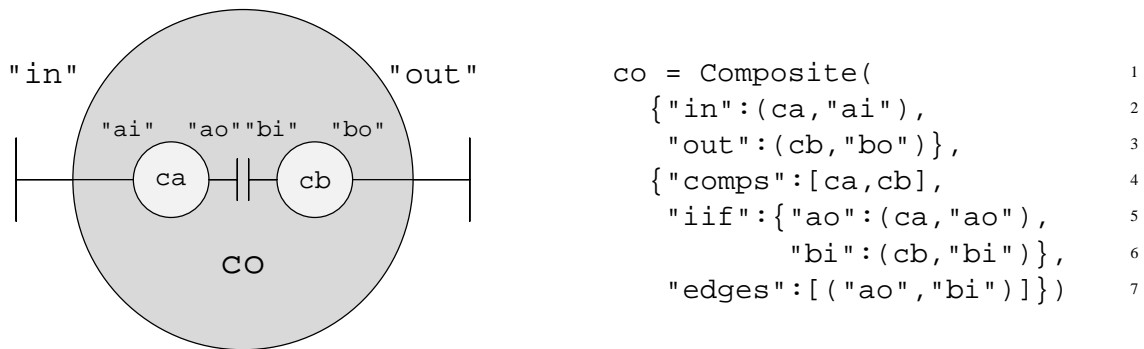


Figure 2: A composite component

3.3 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 [20]. Binding objects for streams and signals are also available in OOPP.

An operational binding can be created with the constructor of the `OpBinding` class. However, since the binding object is usually immediately used to connect two interfaces the `remoteBind` function can be used instead. The difference 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).

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

The stream binding provided 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 with the `streamBind` function. The `streamBind` function creates a stream binding between a stream source interface and a stream sink interface.

4 Infrastructure

The infrastructure is a supporting environment for programs in OOPP. Table 1 describes the different arguments and return values used in the descriptions of the services in the following text.

4.1 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 requests through a capsule proxy are forwarded to the capsule the capsule proxy 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. A capsule proxy for a remote

<code>i, j</code>	Interface references	<code>k</code>	Key or name
<code>l</code>	Local binding control object	<code>m</code>	Method name
<code>c, d</code>	Component instances	<code>f</code>	Method implementation
<code>u</code>	Unique component identifier	<code>a</code>	Argument tuple
<code>C</code>	Class or factory	<code>w</code>	Argument dictionary
<code>p</code>	Capsule proxy	<code>I</code>	Inspect dictionary
<code>r</code>	Result value	<code>x*</code>	<code>x</code> is optional

Table 1: Description of arguments and return values

```

u = p.mkComponent(C)           1           b = remoteBind(i, j)           4
i = p.getIRef(u, "i")         2           r = j.f(2)                   5
j = IRef(None, [], ["f"])    3           r = callMethod(u, "i", "f", (2,)) 6

```

Figure 3: Creating components and calling methods in remote capsules

capsule is created with the `CapsuleProxy` class. The most common services provided by a capsule (and a capsule proxy) are listed in the first column of Table 2.

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 Python code in Figure 3 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`.

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 needed³.

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 code in Figure 3 illustrates the difference between using an operational binding (line 5) and `callMethod` (line 6). Method `f` is in both cases called with an argument 2 and the result is saved in `r`.

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.

³A 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).

Capsule	Name server
<code>registerComponent(c) → u</code>	<code>exportIRef(k, i)</code>
<code>mkComponent(C, a*, w*) → u</code>	<code>exportCaps(k, p)</code>
<code>delComponent(u)</code>	<code>delIRef(k)</code>
<code>getIRef(u, k) → i</code>	<code>delCaps(k)</code>
<code>localBind(i, j) → l</code>	<code>lookupIRef(k) → i</code>
<code>breakBinding(i, j)</code>	<code>importIRef(k) → i</code>
<code>callMethod(u, k, m, a*, w*) → r*</code>	<code>importCaps(k) → p</code>

Table 2: The most common capsule and name server services

4.2 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 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 name server is listening on (a default port is used when it is not explicit given). The name server provides the services listed in the second column of Table 2.

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 is called a proxy and 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 exist because capsules are only accessed through implicit bindings.

4.3 Factories

The task of a factory is to add new components to the environment [21]. An example of a complex factory is a stream binding factory. This factory has to create instances of stub components in different capsules. Other components like buffers and monitors might also have to be added. Some components might have to reserve resources and this could include negotiations. Because of the complexity of this task, factories are often specialised for the needs of a given application or task.

The prototype provides some generic factories for primitive and composite components. The syntax of a call to the generic factory for composite components is complex. It has to contain a complete description of the new composite component. The `remoteBind`, `streamBind` and `sigBind` functions presented above are examples of specialised factories with a simple syntax.

5 Meta-models

OOPP implements two of the Open-ORB meta-models: the encapsulation and the composition meta-model.

5.1 Encapsulation

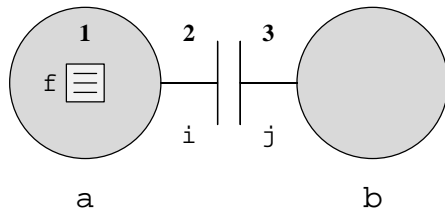
The encapsulation meta-model provides access to the representation or the implementation of interfaces and objects (components). The encapsulation meta-model of a given object or interface is

```

def getX(self):           1           ei = encapsulation(o.i)    6
    return self.x        2           ei.addMethod("get",o.get) 7
eo = encapsulation(o)     3           ej = encapsulation(j)     8
eo.addMethod("get",getX) 4           ej.addImpMethod("get")    9
print o.get()             5           print j.get()            10

```

Figure 4: Adding methods with encapsulation meta-objects



```

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

```

Figure 5: Different locations to manipulate methods

accessed through its encapsulation meta-object. A meta-object does not exist until it is accessed. The `encapsulation` function is used to get access to the encapsulation meta-object of an object or an interface. The most common services of the encapsulation meta-object are listed in the first column of Table 3. The services marked with [†] are only available from the meta-objects of objects and services marked with [‡] are only available from the meta-object of interfaces. The services listed without any marks have effect on the exported methods when they are applied on interfaces.

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 the Python code listed in Figure 4 (line 1–4).

The `addXxxMethod` listed in Table 3 represents the three methods `addPreMethod`, `addPostMethod` and `addWrapMethod` that are used to add pre-, post- and wrap-methods, respectively. A pre-method of a method is another 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 of a method is another 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.

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 in Figure 4 (line 6–9) adds method `get` from object `o` to these interfaces. Notice that `addImpMethod` also updates the local binding.

Figure 5 illustrates the different locations where pre-, post-, and wrap-methods can be added. 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 5). 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 5). 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

Encapsulation meta-model	Composition meta-model
<code>inspect() → I</code>	<code>inspect() → I</code>
<code>addMethod(k, f)</code>	<code>add(c)</code>
<code>addXxxMethod(k, f)</code>	<code>remove(c)</code>
<code>addImpMethod(k)‡</code>	<code>bind(i, j)</code>
<code>addImpXxxMethod(k, f)‡</code>	<code>break(i, j)</code>
<code>changeClass(C)†</code>	<code>replace(c, d)</code>

Table 3: The most common encapsulation and composition meta-object operations

interface `j` (**3** in Figure 5). The Python code in Figure 5 adds a post-method at each location **1** (line 4), **2** (line 5) and **3** (line 6). All the added post-methods print out the return value ("result") before they increase it with one. Suppose method `f` with argument `2` in object `a` returns the value `1`. The output from the Python code in Figure 5 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 example is to monitor when a given attribute becomes greater than a limit value and then raise an alarm.

The `changeClass` method makes it possible to change the class of the object at any time. The consequence is that you change the implementation (code) of a running object.

5.2 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 [22]. The last column of Table 3 lists the operations available from the composition meta-object.

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.

6 QoS management

OOPP includes support for quality of service (QoS) management [23]. 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.

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 too often (with a given definition of 'too often'). A *strategy selector* collects information from the

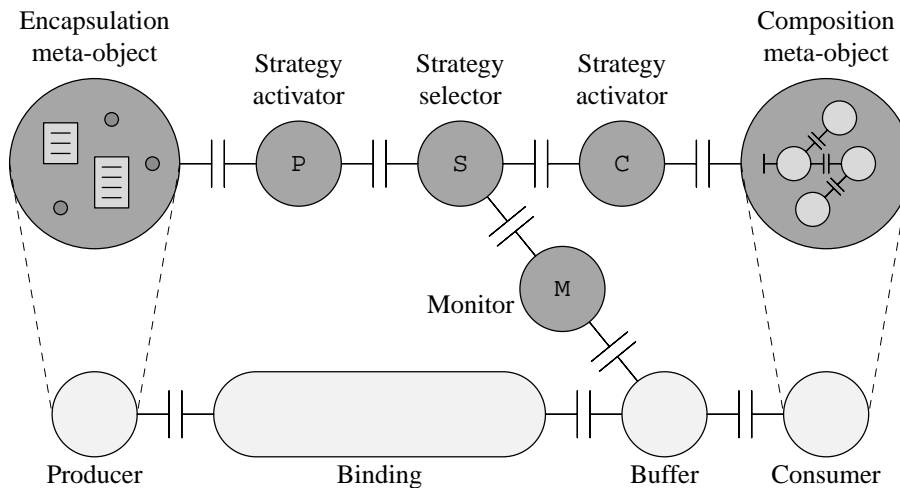


Figure 6: A management setup for the producer/consumer case

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 6 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.

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 [24, 25].

7 Concluding remarks

This paper has presented OOPP, the Open-ORB Python Prototype. OOPP implements a reflective component-based middleware platform with support for quality of service management. The paper focused on the programming model and the meta-models provided by OOPP. OOPP implements an RM-ODP like programming model where the configurability and openness is provided in a principled way via the concept of reflection. Applications including multimedia, real-time requirements and mobile computing seem to benefit from this approach. Deployable components are the foundation for the flexibility (reconfiguration) provided by the system.

Experiences show that OOPP can successfully be used in the development of a wide variety of applications. Performance has not been a primary concern in current OOPP. However, the prototype has been able to deliver the required performance in audio stream examples with reasonable complex quality of service management [26].

The current version of OOPP is available from the Python starship.⁴

⁴<http://starship.python.net/crew/anders/oopp/>

Acknowledgements

Michael Papatomas has contributed a lot to the early attempts to introduce reflection to the programming model. 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. Geoff Coulson has asked the difficult questions and contributed a lot to the development and understanding of the meta-models. 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] P. A. Bernstein, "Middleware: A model for distributed system services," *Communications of the ACM*, vol. 39, pp. 86–98, Feb. 1996.
- [2] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran, N. Parlavantzas, and K. Saikoski, "A principled approach to supporting adaptation in distributed mobile environments," in *5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000)*, (Limerick, Ireland), June 2000.
- [3] F. Eliassen, A. Andersen, G. S. Blair, F. Costa, G. Coulson, V. Goebel, Ø. Hansen, T. Kristensen, T. Plageman, H. O. Rafaelsen, K. B. Saikoski, and W. Yu, "Next generation middleware: Requirements, architecture, and prototypes," in *7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '99)*, (Tunisia, South Africa), Dec. 1999.
- [4] G. S. Blair, G. Coulson, P. Robin, and M. Papatomas, "An architecture for next generation middleware," in *Middleware '98*, Sept. 1998.
- [5] B. C. Smith, *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1982.
- [6] G. Kiczales, J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. The MIT Press, 1991.
- [7] T. Watanabe and A. Yonezawa, "Reflection in an object-oriented concurrent language," in *OOPSLA '88 Proceeding*, vol. 28 of *Sigplan Notices*, pp. 306–315, ACM Press, Sept. 1988.
- [8] G. Agha, "The structure and semantics of actor languages," in *The Proceedings of REX School/Workshop Foundations of Object-Oriented Languages* (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), vol. 489 of *Lecture Notes in Computer Science*, (Noordwijkerhout, The Netherlands), pp. 1–59, Springer-Verlag, May/June 1990.
- [9] Y. Yokote, "The Apertos reflective operating system: The concept and its implementation," in *Proceedings of OOPSLA '92*, vol. 28 of *Sigplan Notices*, pp. 414–434, ACM Press, 1992.
- [10] J. McAffer, "Meta-level architecture support for distributed objects," in *Proceedings of Reflection '96* (G. Kiczales, ed.), (San Francisco), pp. 39–62, 1996.
- [11] C. Szyperski, *Component Software, Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley, 1998.
- [12] ISO/IEC, "Open distributed processing reference model, part 2: Foundations," ITU-T Rec. X.902 — ISO/IEC 10746-2, ISO/IEC, 1995.

- [13] K. B. Saikoski and G. Coulson, "Adaptive groups in Open ORB," in *Proceedings of the CAiSE '99 Doctoral Consortium*, (Heidelberg, Germany), June 1999.
- [14] H. Okamura, Y. Ishikawa, and M. Tokoro, "AL-1/D: A distributed programming system with multi-model reflection framework," in *Proceedings of the Workshop on New Models for Software Architecture*, Nov. 1992.
- [15] G. S. Blair, F. Costa, G. Coulson, H. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, and J.-B. Stefani, "The design of a resource-aware reflective middleware architecture," in *Proceeding of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99)*, vol. 1616 of *Lecture Notes in Computer Science*, (St-Malo, France), pp. 115–134, Springer-Verlag, 1999.
- [16] A. Andersen, G. S. Blair, and F. Eliassen, "A reflective component-based middleware with quality of service management," in *PROMS 2000, Protocols for Multimedia Systems*, (Cra-cow, Poland), Oct. 2000.
- [17] K. Farooqui, L. Logrippo, and J. de Meer, "The ISO reference model for open distributed processing: an introduction," *Computer Networks and ISDN Systems*, vol. 27, pp. 1215–1229, July 1995.
- [18] ISO/IEC, "Open distributed processing reference model, part 1: Overview," ITU-T Rec. X.901 — ISO/IEC 10746-1, ISO/IEC, 1995.
- [19] D. C. Luckham, J. Vera, and S. Meldal, "Three concepts of system architecture," Tech. Rep. CSL-TR-95-674, Computer Systems Laboratory, Stanford University, July 1995.
- [20] ISO/IEC, "Open distributed processing reference model, part 3: Architecture," ITU-T Rec. X.903 — ISO/IEC 10746-3, ISO/IEC, 1995.
- [21] G. Blair and J.-B. Stefani, *Open Distributed Processing and Multimedia*. Addison-Wesley, 1998.
- [22] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. 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.
- [23] G. Blair, A. Andersen, L. Blair, G. Coulson, and D. S. Gancedo, "Supporting dynamic QoS management functions in a reflective middleware platform," *IEE Proceedings – Software*, 2000.
- [24] L. Blair, *The Formal Specification and Verification of Distributed Multimedia Systems*. Dr. thesis, Department of Computer Science, Lancaster University, Sept. 1994.
- [25] G. Blair, L. Blair, H. Bowman, and A. Chetwynd, *Formal Specification of Distributed Multimedia Systems*. UCL Press, 1998.
- [26] D. S. Gancedo, "QoS MonAuTA, QoS monitoring and adaptation using timed automata," Master's thesis, Lancaster University, UK, Sept. 1999.