Appendix C

Compendium

In «A Reflective Component-Based Middleware with Quality of Service Management» (se Section C.1 below) OOPP is presented as reflective middleware with the possibility to inspect, adapt and extend the components of the system to satisfy the requirements of a given application. Quality of service management is achieved with management components with different roles (monitors, strategy selectors and strategy activators). Timed automata is presented as one way of specifying the behaviour of such management components. The paper has been published in the Proceedings of the 5th International Conference on Protocols for Multimedia Systems (PROMS 2000, Cracow, Poland, October 2000).

In «Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform» (se Section C.2 below) the role of reflection in supporting the dynamic QoS management functions of monitoring and adaptation is considered. It is argued that reflection provides strong support for such functions and, indeed, the approach offers important benefits over alternative implementation strategies. The paper has been published in IEE Proceedings – Software (Volume 147, Issue 1, June 2000)[1].

C.1 Proceedings PROMS 2000

A REFLECTIVE COMPONENT-BASED MIDDLEWARE WITH QUALITY OF SERVICE MANAGEMENT

Anders Andersen[1] *Gordon S. Blair*[2,1] *Frank Eliassen*[3,1]
[1]Department of Computer Science, University of Tromsø,
N-9037 Tromsø, Norway, e-mail: aa@computer.org
[2]Computing Department, Lancaster University,
Lancaster LA1 4YR, UK, e-mail: gordon@comp.lancs.ac.uk
[3]Department of Informatics, University of Oslo,
N-0316 Oslo, Norway, e-mail: frank@ifi.uio.no

ABSTRACT

The role of middleware is to present a unified programming model to application writers and mask out the problems of heterogeneity and distribution. However, new application types need support for multimedia, real-time and mobility. This support should be configurable to satisfy the requirements from a wide variety of applications. But this is not enough. Mobility and continuous media requires 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 open engineering. The Open-ORB Python Prototype (OOPP) presented in this paper implements a reflective middleware with the possibility to inspect, adapt and extend the components of the system to satisfy the requirements of a given application.

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, 2, 3]. 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) increasing mobility. 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 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, 5]. 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 meta-models. Section 6 discusses QoS management in the implemented prototype. Section 7 gives an overview over related work and section 8 presents some ideas for future work. Section 9 concludes the paper.

2 OPEN-ORB

2.1 Reflective middleware

The reflection hypothesis introduced by Smith in 1982 [6] 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 [7, 8, 9]. Later reflection has been applied to operating systems [10] and, more recently, distributed systems [11].

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 [12]. 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 [13] where (i) components have interfaces, (ii) interfaces for continuous media are supported and (iii) explicit bindings can be created between compatible interfaces.

In addition, Open-ORB meta-level 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 [14].

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 [15]. 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 [8]. 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 of a composite component describes the contained components and how these components are related (connected). The component graph can be inspected and altered using the composition meta-model of an composite component. All interfaces of a component provides the same composition meta-model.

Behavioural reflection is concerned with activity in the underlying system [8]. This is represented by the environmental meta-model. In terms of middleware, the underlying activity equates to functions such as message arrival, enqueing, selection, unmarshalling and dispatching (plus the equivalent on the sending side) [8, 11].

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 [16, 17] 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 [18, 19]. The Open-ORB Python Prototype (OOPP) is influenced by the Open Distributed Processing Reference Model (RM-ODP) [20, 21]. In addition, it provides structural reflection through the encapsulation and composition meta-models. The purpose of OOPP has been to validate the Open-ORB architecture by illustrating the utility of its various adaptation mechanisms.

Python has been selected as implementation language for several reasons. The most important reason is the existing reflective features of the languages [22]. The structural meta-models have proven to be relatively easy to build on top of these reflective features. Another argument for choosing Python is its interpreted nature and flexible typing that makes it ideal for prototyping. The good support for distributed programming is also preferable.

3 PROGRAMMING STRUCTURES

3.1 Interfaces and local bindings

An interface of an object defines a subset of the interactions of that object [13]. 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 methods of one interface to the imported methods of another interface and vice versa. The two objects `a` and `b` in Figure 1 have an interfaces `i` and an interface `j`, respectively. Inter-

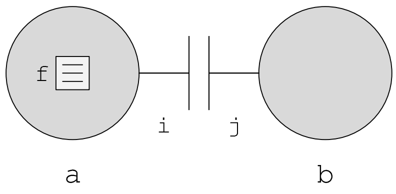

Figure 1. A local binding

face `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 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.[1] 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 and is not dependent on its local binding control object (the local binding even exists after its control object is garbage collected). 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()
```

An interface is accessed through an interface reference created with the `IRef` class. The interface reference contains a description of the interface including (i) its object, (ii) its exported methods, and (iii) its 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 is also 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)
```

[1] 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 CORBA IDL.

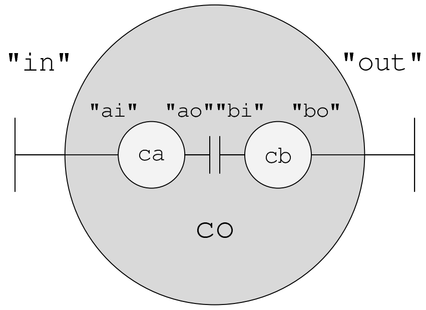

Figure 2. A composite component

A stream interface uses a `put` method with one data argument to insert data (a frame) into the stream. Object `b` in the example above implements the `put` method to receive data from the stream. A signal interface pair is created with the `SigSrcIRef` and the `SigSinkIRef` classes. The signal source interface provides (exports) an `event` method with no arguments for the actual transfer of a signal.

3.2 Components

Components are used to build both applications and the middleware in OOPP. A component is a unit of independent deployment [12]. 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.

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 following statement:

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

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[2] of the component. Interface `"op"` of `ca` (represented by the interface reference `i`) can be accessed with the `ca.interfaces["op"]` expression.

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

[2] The `interface` attribute is a Python dictionary. A dictionary is similar to an associative array in Perl. The expression `{"op":i}` is a dictionary with one element with the key `"op"` and the value `i`.

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` in Figure 2 was created with the following statement:

```
co = Composite(
  {"in":(ca,"ai"),
   "out":(cb,"bo")},
  {"comps":[ca,cb],
   "iif":{"ao":(ca,"ao"),
          "bi":(cb,"bi")},
   "edges":[("ao","bi")]})
```

The first argument specifies the external interfaces of the component and the second the component graph.[3] 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 result is a fairly complex constructor syntax. A composite component class is usually a refinement of the `Composite` class with a less complex constructor syntax. The binding objects discussed below are examples of such composite components.

3.3 Binding objects

Local bindings can only be used between interfaces in the same address space. In contrast, binding objects can be used to create bindings between interfaces in different address spaces or even on different nodes. A binding object replaces a local binding in such cases. An operational binding is a particular kind of binding object supporting operational interactions [23]. Binding objects for streams and signals are also available in OOPP.

Figure 3 illustrates how the built-in operational binding of OOPP is implemented. Capsule `A` and capsule `B` are two different address spaces. The operational binding is a composite component containing two components, `stub1` and `stub2`. It provides three external interfaces, one control interface `"ctrl"` and two connecting interfaces `"iface1"` and `"iface2"`. The two stubs are connected with two low-level TCP/IP bindings. One is for requests (method calls) from an object in capsule `A` to an object in capsule `B` (with reply back to the object in capsule `A`) and the other one is for requests from an object in capsule `B` to an object in capsule `A` (also with reply). Operational bindings are thus two-way bindings (method calls can go in both directions). The low-level TCP/IP binding are implemented with an `m` and an `l` object. The `m` objects are message objects (send request and receive reply) and the `l` objects are listen objects (receive request and send reply).

[3] In Python, dictionaries are written with curly braces, lists with square brackets, and tuples with normal parentheses. Elements are separated with commas.

An operational binding can be created with the constructor of the `OpBinding` class. However, since the binding 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). 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)
r = j.f(2)
```

The built-in signal binding provided by OOPP is also based on TCP/IP. 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 by OOPP is based on connected UDP/IP. It 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. 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 or with the `streamBind` function. 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)
for i in range(100):
    src.put(data[i])
```

4 INFRASTRUCTURE

The infrastructure is a supporting environment for programs in OOPP. The infrastructure is influenced by the engineering viewpoint of RM-ODP [23].

Table 1 describes the different arguments and return values used in the descriptions of the services in the following text.

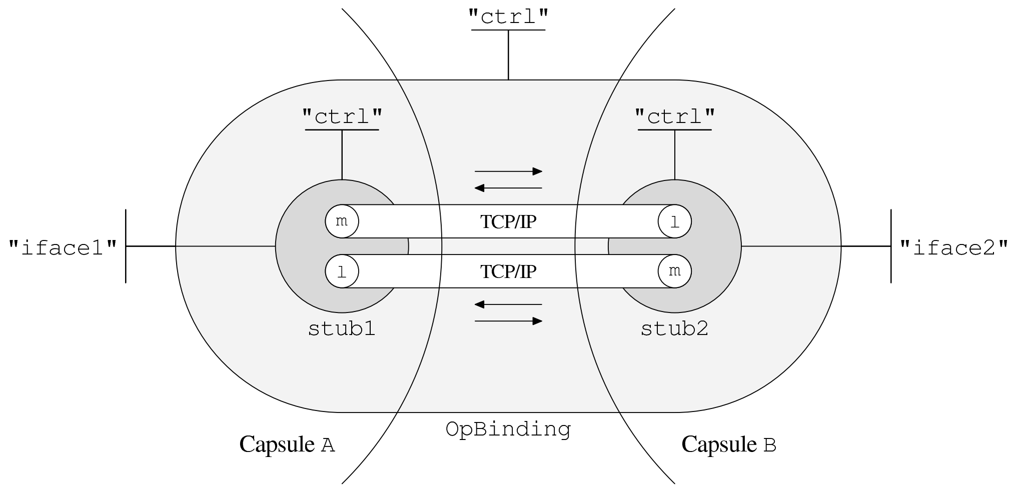

Figure 3. The implementation of an operational binding

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

Table 1. Description of arguments and return values

4.1 Capsules

An OOPP address space is called a capsule. A capsule provides services for its local components (the components located in the address space). 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 it represents and replies are returned back to the caller through the proxy. The local capsule services is available through the `local` attribute of the `capsule` module. A capsule proxy for a remote capsule is created with the `CapsuleProxy` class. The following services represent the most common services provided by a capsule (and a capsule proxy):

```
registerComponent(c)→u
mkComponent(C,a*,w*)→u
delComponent(u)
getIRef(u,k)→i
localBind(i,j)→l
breakBinding(i,j)
callMethod(u,k,m,a*,w*)→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)
i = p.getIRef(u,"i")
b = remoteBind(i,j)
r = j.f(2)
```

The `getIRef` service of a capsule returns a global interface reference. In the example above the global interface reference of interface `"i"` in the remote component `u` is returned. The `getIRef` service of a capsule should always be used when a global interface reference of a registered component is needed.[4]

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.

[4] 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).

The `callMethod` service of a 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`:

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

The capsule also provides other low-level methods not listed above. This includes announcements (method calls without any return value), asynchronous (non-blocking) method calls and services to duplicate and move components. A capsule proxy can only be used to access a capsule if the capsule has started its serving loop. This has to be done explicitly with the `serve` method of the local capsule.

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 (exported) and looked-up (imported) by a key (name). 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).

4.3 Factories

The task of a factory is to add new components to the environment [24]. 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 system.

OOPP provides some generic factories for primitive and composite components. The syntax of 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 simpler syntax.

5 META-MODELS

OOPP implements two of the Open-ORB meta-models: the encapsulation and the composition meta-model. An implementation of the resource meta-model is presented in [17] and [25].

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 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. This is the most common services of the encapsulation meta-object:

```
inspect()→I
addMethod(k,f)
addXxxMethod(k,f)
addGetAttr(k,f)†
addSetAttr(k,f)†
changeClass(C)†
addImpMethod(k)‡
addImpXxxMethod(k,f)‡
changeObject(n)‡
restore()
```

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.

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 the following code:

```
def getx(self):
    return self.x
eo = encapsulation(o)
eo.addMethod("get",getx)
print o.get()
```

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

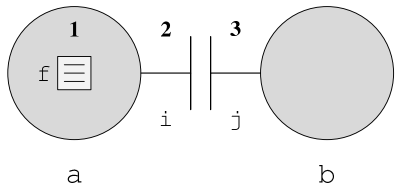

Figure 4. Different places to manipulate methods

can manipulate the arguments and the return values. It can even decide not to call the actual method at all. 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):
    print m["result"]
eo = encapsulation(o)
eo.addPostMethod("get",p)
```

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 object (as in every method of an object in Python). The second argument `m` is a dictionary containing information about the method call including 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 methods 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)
ei.addMethod("get",o.get)
ej = encapsulation(j)
ej.addImpMethod("get")
print j.get()
```

Figure 4 illustrates the different places 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 4). 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 4). 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 4). The example below adds a post-method at each place **1** (line 4), **2** (line 5) and **3** (line 6) in Figure 4. All the added post-methods print out the return value (`"result"`) before they increase it with one.

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

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 range checks and different monitoring tasks. One possibility is to monitor when a given attribute becomes greater than a limit value and then raise an alarm.

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 [26]. The following operations are available from the composition meta-object:

```
inspect()→I
add(c)
remove(c)
bind(i,j)
break(i,j)
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.

6 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.

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 5 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 [27, 28].

An example of the usage of this QoS management approach is presented in [29]. In this example an audio stream dynamically adapts to the variable QoS delivered by the network. This could be further extended with resource management using the resource meta-model presented in [17].

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

7 RELATED WORK

There is growing interest in the use of reflection in distributed systems. Pioneering work in this general area was carried out by McAffer [11]. With respect to middleware in particular, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) [30, 31], focusing on reification of invocation marshalling and dispatching. In addition, researchers at APM have developed an experimental middleware platform called FlexiNet [32], which 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. Researchers at DSTC have developed a distributed infrastructure for mobile computing [33]. This infrastructure includes adaption rules to sense and react to environment variations. Manola has carried out work in the design of a "RISC" object model for distributed computing [34], i.e. a minimal object model which can be specialised through reflection. Finally, researchers at the Ecole des Mines de Nantes are also investigating the use of reflection in proxy mechanisms for ORBs [35].

Our design has been influenced by a number of specific reflective languages. 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 [15]. Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R [8] and CodA [11]. 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.

8 FUTURE WORK

OOPP currently only implements the encapsulation and the composition meta-models. One obvious task for future work is to implement the environment and the resource meta-models.

An implementation of the environment meta-model could learn a lot from ATOM [36]. The separation of the concurrent execution and synchronisation features from the object-oriented features in the ATOM programming model has a lot in common with the distinct environment meta-model in Open-ORB. The implementation of the resource meta-model should be based on the work presented in [17] and [25].

A tool to manipulate and deploy the components of an application in a graphical user interface (GUI) would increase the power of this programming environment. Such a tool should also have the possibility to access the meta models of the interfaces and components.

Other future tasks include the implementation of a generic event notification service for components and a richer set of (stream) binding objects, and adding type information and type checks for the interfaces. Security and performance are other obvious tasks ignored so far.

9 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

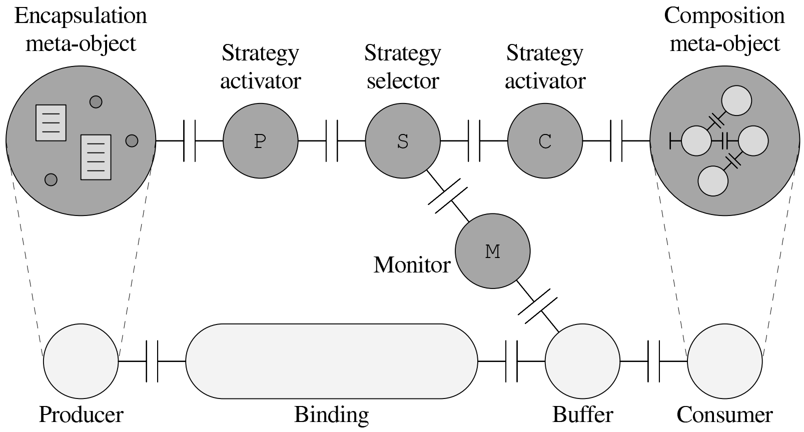

Figure 5. A management setup for the producer/consumer case

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 [29].

The current version of OOPP is available from the Python starship.[5]

ACKNOWLEDGEMENTS

Michael Papathomas 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, thank you to the various members of the Distributed Multimedia Research Group at Lancaster University, the Information and Communication Technology group at NORUT IT, and the Department of Computer Science at University of Tromsø 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] W. W. Eckerson, "Three-tier client/server architecture," *Open Information Systems*, vol. 10, pp. 3–22, Jan. 1995.

[3] S. S. King, "Middleware!," *Data Communications*, vol. 21, pp. 58–67, Mar. 1992.

[4] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An architecture for next generation middleware," in *Middleware'98*, Sept. 1998.

[5] F. Costa, G. Blair, and G. Coulson, "Experiments with reflective middleware," in *ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems*, vol. 1543 of *Lecture Notes in Computer Science*, (Brussels, Belgium), pp. 390–391, July 1998.

[6] B. C. Smith, *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[7] G. Kiczales, J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[8] 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.

[9] 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.

[10] Y. Yokote, "The Apertos reflective operating system: The concept and its implementation," in *Proceedingss of OOPSLA'92*, vol. 28 of *Sigplan Notices*, pp. 414–434, ACM Press, 1992.

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

[11] J. McAffer, "Meta-level architecture support for distributed objects," in *Proceedings of Reflection'96* (G. Kiczales, ed.), (San Francisco), pp. 39–62, 1996.

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

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

[14] K. B. Saikoski and G. Coulson, "Adaptive groups in Open ORB," in *Proceedigns of the CAiSE '99 Doctoral Consortium*, (Heidelberg, Germany), June 1999.

[15] 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.

[16] 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.

[17] H. A. Duran and G. Blair, "A resource management framework for adaptive middleware," in *3th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, (Newport Beach, California, USA), Mar. 2000.

[18] G. van Rossum, "Glue it all together with Python," in *Workshop on Compositional Software Architectures*, (Monterey, California), Jan. 1998. (position paper).

[19] M. Lutz, *Programming Python*. A Nutshell Handbook, O'Reilly & Associates, Inc., 1996.

[20] 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.

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

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

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

[24] G. Blair and J.-B. Stefani, *Open Distributed Processing and Multimedia*. Addison-Wesley, 1998.

[25] H. A. Duran and G. Blair, "Configuring and reconfiguring resources in middleware," in *1st International Symposium on Advanced Distributed Systems*, (Guadalajara, Jalisco, Mexico), Mar. 2000.

[26] 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.

[27] L. Blair, *The Formal Specification and Verification of Distributed Multimedia Systems*. Dr. thesis, Department of Computer Science, Lancaster University, Sept. 1994.

[28] G. Blair, L. Blair, H. Bowman, and A. Chetwynd, *Formal Specification of Distributed Multimedia Systems*. UCL Press, 1998.

[29] D. S. Gancedo, "QoSMonAuTA, QoS monitoring and adaptation using timed automata," Master's thesis, Lancaster University, UK, Sept. 1999.

[30] A. Singhai, A. Sane, and R. Campbell, "Reflective ORBs: Supporting robust, time-critical distribution," in *ECOOP'97 Workshops — Reflective Real-Time Object-Oriented Programming and Systems* (J. Bosch and S. Mitchell, eds.), vol. 1357 of *Lecture Notes in Computer Science*, (Jyväskylä, Finland), pp. 55–61, Springer-Verlag, June 1997.

[31] M. Roman, F. Kon, and R. H. Campbell, "Design and implementation of runtime reflection in communication middleware: the dynamicTAO case," in *ICDCS'99 Workshop on Middleware*, (Austin, Texas), May 1999.

[32] R. Hayton, "FlexiNet open ORB framework," Tech. Rep. APM.2047.01.00, ANSA Phase III, APM, 1997.

[33] A. Rakotonirainy and M. Chilvers, "A distributed infrastructure for mobile computing," in *The Sixth International Python Conference*, (San Jose, California), pp. 89–95, Oct. 1997.

[34] F. Manola, "Meta object protocol concepts for a "RISC" object model," Tech. Rep. TR-0244-12-93-165, GTE Laboratories, 1993.

[35] T. Ledoux, "Implementing proxy objects in a reflective ORB," in *Procceings of the ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation*, (Jyväskylä), 1997.

[36] M. Papathomas and A. Andersen, "Concurrent object-orieted programming in Python with ATOM," in *The Sixth International Python Conference*, (San Jose, California), pp. 77–87, Oct. 1997.

C.2 IEE Proceedings – Software 147(1)

Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform

Gordon S. Blair, Anders Andersen*, Lynne Blair, Geoff Coulson and David Sánchez**

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

E-mail: [gordon, lb, geoff]@comp.lancs.ac.uk

* aa@computer.org

** david.sanchez@switzerland.org

Abstract

Reflection has recently been applied in a variety of settings to introduce more openness and flexibility into designs. This paper builds on our previous work in applying reflection to the design of middleware platforms by considering the role of reflection in supporting the dynamic QoS management functions of monitoring and adaptation. It is argued that reflection provides strong support for such functions and, indeed, the approach offers important benefits over alternative implementation strategies. A pilot implementation of our QoS management scheme is discussed and evaluated and examples of the use of our approach are given.

1. INTRODUCTION

Reflection initially emerged in the programming language community as an important technique to introduce more flexibility and openness into the design of programming languages. In recent years, researchers have studied the potential impact of reflection in other areas including windowing systems, operating systems and file systems. Again, the motivation has been to introduce more flexibility and openness in the designs. The authors believe that the same technique can usefully be exploited in the area of distributed systems, in general, and middleware[1], in particular. For example, reflection can help provide the necessary level of adaptability required by emerging distributed systems application areas such as multimedia and mobile computing.

In previous papers, the authors have reported on the design and implementation of reflective middleware platforms [Blair98, Costa98]. The aim of this paper is to consider the implications of such an architecture for the area of Quality of Service (QoS) management. More specifically, the paper describes how critical QoS management functions can be incorporated into our reflective middleware platform, with particular emphasis on the *dynamic* QoS management functions of QoS monitoring and adaptation. It is argued that the reflective approach leads to a natural and highly flexible implementation of such QoS management functions (and indeed the approach offers considerable benefits over more conventional approaches).

The paper is structured as follows. Section 2 presents the case for reflective middleware and section 3 summarises the main features of our architecture. Section 4 then examines our approach to

---

[1] We use the term *middleware* to refer to distributed systems platforms that offer a language and platform independent programming model, located between the application and the underlying distributed environment. Examples of middleware platforms include the OMG's CORBA [OMG98a] and Microsoft's DCOM.

QoS management, highlighting the role of timed automata in realising key QoS management functions. Consideration is also given to the issue of formal verification of QoS management subsystems. Following this, section 5 discusses and evaluates our pilot implementation and presents examples of our approach, illustrating the role of reflection in supporting QoS management policies. Finally, section 6 discusses related work and section 7 presents some concluding remarks.

2. THE CASE FOR REFLECTIVE MIDDLEWARE

The concept of reflection was first introduced by Smith in 1982 [Smith82]. 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 operations 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 [Kiczales91, Watanabe88, Agha91]. More recently, the work has diversified with reflection being applied in operating systems [Yokote92] and, more recently, distributed systems (see section 6).

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 to reduce the bandwidth requirements of a communications stream. Although reflection is a promising technique, there are a number of potential drawbacks of this approach; in particular issues of *integrity* and *performance* must be carefully addressed (we return to these issues in sections 3.3 and 7 respectively).

In contrast, the present-day approach to developing middleware platforms is generally to adopt a *black box* philosophy, whereby implementation details are hidden from the platform user (cf. distribution transparency). 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. The recently defined Portable Object Adapter is another attempt to introduce more openness in their design. Nevertheless, their overall approach can be criticised for being rather ad hoc. Similarly, a number of ORB vendors have felt obliged to expose selected aspects of the underlying system (e.g. filters in Orbix or interceptors in COOL). These are however non-standard and hence compromise the portability of CORBA applications and services. The RM-ODP architecture partially addresses this issue by distinguishing between computational and engineering concerns. However, the authors believe this approach is still not sufficient to meet the demands of the next generation of distributed applications (see section 3.3 below).

The authors believe that the solution is to provide flexible middleware platforms through application of the principle of reflection.

3. AN ARCHITECTURE FOR REFLECTIVE MIDDLEWARE

3.1. General Principles

In our reflective architecture, we adopt a *component-based* model of computation [Szyperski98]. A middleware platform is then viewed as a particular configuration of components, which can be selected at build-time and re-configured at run-time. We therefore provide an open and extensible library of components, and component factories, supporting the construction of such platforms, e.g. protocol components, schedulers, etc. The use of components is important given the trend towards the application of this technology in open distributed processing, e.g. CORBA v3 [OMG99] and Microsoft's DCOM. Note however that these technologies exploit component technology at the application level; we extend this approach to the structuring of the middleware platform itself. Our particular component model includes features to support multimedia applications, and is derived from previous work on the Computational Model from RM-ODP. The main features of our component model are: i) components are described in terms of a set of *required* and *provided* interfaces, ii) interfaces for *continuous media* interaction are supported, iii) *explicit bindings* can be created between compatible interfaces (the result being the creation of a *binding component*), and iv) components offer a built-in *event notification service*. The component model also has a sophisticated model of quality of service including QoS annotation on interfaces. Further details on the underlying RM-ODP Computational Model can be found in [Blair97]. In contrast, with RM-ODP, however, we adopt a consistent computational model throughout the design (see section 3.3).

As our second principle, we adopt a *procedural* (as opposed to a declarative) approach to reflection, i.e. the meta-level (selectively) exposes the actual "program" (in our case, configuration of components) that implements the system. In our context, this approach has a number of advantages over the declarative approach. For example, the procedural approach is more primitive (in particular, it is possible to support declarative interfaces on top of procedural reflection but not vice versa). Procedural reflection also opens the possibility of an infinite tower of reflection (i.e. the base level has a meta-level, the meta-level is implemented using components and hence has a meta-meta-level, and so on). This is realised in our approach by allowing such an infinite structure to exist *in principle* 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 [Watanabe88]). Access to different meta-levels is important in our design although in practice most access will be restricted to the meta- and meta-meta-levels (e.g. see the examples in section 5.2).

The third principle underlying our design is to support *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 (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 components in one single action; to support this, we also allow the use of *groups* [Saikoski99] (we do not address this aspect further in this paper however).

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 [Okamura92]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects. Further details of each of the various models can be found below.

3.2. The Design of Meta-space

3.2.1. Supporting Structural Reflection

In reflective systems, structural reflection is concerned with the content of a given component [Watanabe88]. In our architecture, this aspect of meta-space is represented by two distinct meta-models, namely the *encapsulation* and *composition* meta-models. We introduce each model in turn below.

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 (where available). This is equivalent to the introspection facilities available, for example, in the Java language, although we go further by also supporting adaptation. Clearly, however, the level of access provided by the encapsulation model will be language dependent. For example, with compiled languages such as C, access may be limited to introspection of the associated IDL interface. With more open (interpreted) languages, such as Java or Python (see section 5.1), 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 encapsulation 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.

In reality, many components will in fact be *composite*, using a number of other components in their construction. In recognition of this fact, we also provide a *compositional meta-model* offering access to such constituent components. Note that this meta-model is associated with each component and not each interface. More specifically, the same compositional meta-model will be reached from each interface defined on the component (reflecting the fact that it is the component itself, rather than the interface, that is composite). In the meta-model, the composition of a component is represented as a *component graph*, in which the constituent components are connected together by *local bindings*[2]. The interface offered by the meta-model then supports operations to inspect and adapt the graph structure, i.e. to view the structure of the graph, to access individual components in the graph, and to adapt the graph structure and content.

This meta-model is particularly useful when dealing with binding components [Blair98]. In this context, the composition meta-model reifies the internal structure of the binding in terms of the components used to realise the end-to-end communication path. For example the component graph could feature an MPEG compressor and decompressor and an RTP binding component. The structure can also be exposed recursively; for example, the composition meta-model of the RTP binding might expose the peer protocol entities for RTP and also the underlying UDP/IP protocol. It is argued in [Fitzpatrick98] that open bindings alone provide strong support for mobile computing.

2 The RM-ODP inspired concept of local binding is crucial in our design, providing a language-independent means of implementing the interaction point between interfaces.

3.2.2. Supporting Behavioural Reflection

Behavioural reflection is concerned with activity in the underlying system [Watanabe88]. This is represented by a single meta-model associated with each interface, 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) [Watanabe88, McAffer96].

Again, different levels of access are supported. For example, a simple meta-model may enable the insertion of pre- and post- methods. Such a mechanism can be used to introduce, for example, additional levels of distribution transparency (such as concurrency control) or to insert functions such as security managers or compression components. A more complex meta-model may allow the inspection or adaptation of each element of the processing of messages as described above. With such complexity, it is likely that such a meta-model would itself be a composite component with individual components accessed via the compositional meta-space of the environmental meta-space. As with the other meta-models, heterogeneity is accommodated within an open and extensible type hierarchy.

3.2.3. Supporting Resource Access

Most reflective languages and systems restrict their scope to the above styles of reflection. In experiments, however, we have identified a significant weakness of this approach, namely that we have no means of accessing the level of resources and resource management in the system. This is a particular problem for mobile, multimedia and real-time systems where it is often important to be aware of the resources currently available at a given node (e.g. if it is intended to introduce a new software compression component). We therefore introduce a fourth meta-model, referred to as the *resource meta-model* [Blair99b]. This meta-model supports the reification of resource creation, scheduling and, more generally, management. It is important to stress that this meta-model is fully orthogonal to the others. For example, the environmental meta-model identifies the steps involved in processing an arriving message; the resources meta-model then identifies the resources required to perform this processing. More generally, the resources meta-model is concerned with the allocation and management of resources associated with any activity in the system.

The meta-model is based around the abstractions of *resources* and *tasks*[3]. Resources can be either primitive (e.g. raw memory or OS threads) or complex (e.g. buffers or user-level threads multiplexed on OS threads). They are created and managed by *resource factories* and *resource managers* which typically build complex resources by adding value to, or combining, primitive resource instances. For example, a user level scheduler is a resource manager which builds user level threads from OS threads. Both resources and their managers are uniformly viewed as components. Tasks are then the logical unit of activity in the system with the precise granularity varying from configuration to configuration. For example, there could be a single task dealing with the arrival, filtering and presentation of an incoming video stream, or alternatively this could be divided into a number of smaller tasks. Importantly, tasks can span component boundaries and are thus orthogonal to the structure of the system. Tasks are essentially the unit of resource allocation, i.e. tasks have a pool of resources to carry out their desired activity. Tasks also have associated task managers (analogous to the services identified above for resources).

There is a resources meta-model *per address space*, i.e. resources are associated with a particular address space and all components within that address space share the same meta-model. The meta-model provides access to a set of components representing resources, together with the associated managers. As with other meta-models, it is then possible to either inspect or adapt activity associated with resources. For example, it is possible to insert monitors to capture statistics on the effectiveness of a thread scheduling policy and then possibly change this policy based on the information collected. In programming terms, the resources meta-model is accessed in a similar way to the compositional meta-model; i.e. as a graph structure which organises resources, tasks and managers into hierarchical structures.

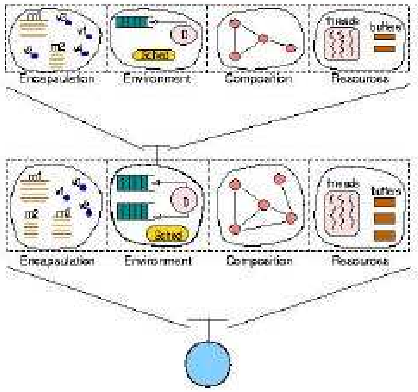

Figure 1: The structure of meta-space.

The complete architecture is summarised in figure 1.

3.3. Discussion

In our opinion, the reflective architecture described above provides a promising basis for the design of future middleware platforms and overcomes certain 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. 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, this approach enforces a two-level structure, i.e. it is not possible to analyse engineering objects in terms of their internal structure

[3] This aspect of the work is being developed along with our collaborators at CNET, France Telecom. In particular, they have developed the particular resource/ task model presented in this section.

or behaviour. Our approach overcomes this limitation by offering a consistent object (component) model throughout, supporting arbitrary levels of openness.

Another benefit of our approach is that it reduces problems of maintaining integrity. This is due to our approach to scoping whereby every component/interface has its own meta-space. Thus changes to a meta-space can only affect a single component. Furthermore, the meta-space is highly structured, again minimising the scope of changes. An additional level of safety is provided by the strongly typed component model.

4. INTRODUCING QOS MANAGEMENT

4.1. General Approach

As can be seen from the descriptions above, the concept of component graphs is central to our design. In particular, the composition, environment and resource meta-models are all represented as component graphs. Given this, QoS management is largely concerned with:

i) creating the initial configurations of components and resources based on a statement of desired QoS properties and the assumed QoS from the environment (e.g. selecting appropriate compression components to be used), and

ii) inspecting and adapting the corresponding component graphs depending on the actual QoS attained and the actual QoS offered by the environment (which might of course change in, for example, a mobile environment).

The first aspect is referred to as *static* QoS management and the second aspect as *dynamic* QoS management. As stated earlier, we focus our attention on the dynamic aspects of QoS management in this paper (the static aspects are the responsibility of the various factories, including binding factories, and are the subject of ongoing research). Dynamic QoS management is achieved by introducing *management components* into the component graph structure (accessed via meta-space). To maintain a clean separation of concerns between management components and components being managed, communication between the two is achieved by an *event notification* mechanism (as provided by our component model; see section 3.1). Separation is achieved because components do not need to know in advance if they are to be managed. In other words, managers can be introduced at any time and can then register for events of interest (and receive call-backs when the specific events occur).

Different styles of management component are identified in our architecture (see table 1).

Monitoring	
Event Collector	Observe behaviour of underlying functional components and generate relevant QoS events.
Monitor	Collect QoS events and report abnormal behaviour to interested parties.
Control	
Strategy Selectors	Select an appropriate adaptation strategy (i.e. strategy activator) based on feedback from monitors.
Strategy Activators	Implement a particular strategy, e.g. by manipulating component graph.

Table 1: Styles of management component.

The role of *monitoring* is to collect statistics on the level of QoS attained by the running system and to raise events when problems occur, e.g. to collect information on the latency and throughput of a video presentation and raise exceptions should they fall outside given thresholds. This is sub-divided into event collectors, which interface with the underlying implementation, and monitors which detect QoS violations. *Control* is then responsible for implementing adaptation policies in response to such events. We distinguish between *strategy selectors* and *strategy activators* (which together realise the adaptation policy). Strategy selectors decide on which approach should be taken in response to QoS degradation, e.g. degrading the quality of the video presentation or providing additional resources. In contrast, strategy activators are responsible for the detailed implementation of this strategy, typically by manipulating the component graph, e.g. introducing a jitter compensation buffer or accessing and adapting resources through the resource meta-model. The rationale for this division is to provide a cleaner architecture and to promote re-use of management components. In addition, it is useful to distinguish between the implementation-oriented aspects of QoS management (i.e. event collectors and strategy activators), and the policy-oriented aspects (i.e. monitors and strategy selectors), and perhaps use a different languages for each aspect (see section 4.2.1 below). It should be stressed though that, in implementation, the different functions can be composed together (we expand this aspect of composition in section 4.2.2 below).

At a first glance, this might appear to be a fairly traditional approach to QoS management. However, when combined with the capabilities of a reflective architecture, some interesting properties emerge. Firstly, the approach is completely dynamic. New management components can be introduced at any time and at any place in the underlying configuration. Similarly, they can be removed when no longer needed. Both these actions are initiated by re-configuring component graphs. Secondly, the policy for management is itself open to inspection and adaptation through reification of management components. To enable this, we assume that management components consist of a policy written in an appropriate scripting language, together with an in-built interpreter for that scripting language (see section 5.2).

The architecture is open in that any scripting language can be used, although we do provide support for one particular language (see section 4.2). Reflection can then be used to access or modify this policy, e.g. by downloading a new management script or altering some parameters for the script. More specifically, we can introduce meta-managers (again consisting of monitors and controllers) to effectively manage the management structure. Whereas managers implement policy, meta-managers implement meta-policy. As an example, consider the use of header compression in a low bandwidth environment. A manager could have the role of switching header compression on or off based on a given bandwidth threshold (an attribute of the manager component). A meta-manager would then be able to alter this threshold depending on the current processor load. We argue that this is a useful facility in highly dynamic environments.

4.2. QoS Management Policies

4.2.1. Specification of Policies

As stated above, the architecture is open in that management components can be written using any scripting language. We do however provide support for one particular notation for the specification of QoS management policies, namely *timed automata*. We typically use timed automata for the specification of monitors and strategy selectors, but not necessarily for event collectors and strategy activators (especially, for example, for complex strategies requiring significant manipulation of the component graph structure).

We define our timed automata formally as:

$$TA = \langle S, s_0, \rightarrow, I \rangle$$

where S is a finite set of states,
$s_0 \in S$ is the initial state,
$\rightarrow$ is a transition relation, and
I is an invariant assignment function on a given state; all variables must satisfy the invariant whilst the automaton is operating in this state.

Transition relations have the form:

$$\rightarrow = \langle l, g, a, r, l' \rangle$$

where l, l' are nodes of the automaton,
a are actions,
g is a guard, and
r is a reset (or reassignment) function.

The transitions must satisfy certain rules detailed in [Blair99a]; these only allow transitions to occur when guards are true and ensure that the required variables are reset on transitions. Furthermore, time can only pass in a given state if the invariant at the new time holds true.

Examples of using timed automata can be found in section 5.2 below.

4.2.2. Formal Verification

One benefit of adopting timed automata is that it is then possible to *verify* QoS management functions. Indeed, we can go further and verify QoS management subsystems plus their interactions with the rest of the system. To support this, we are developing a *multi-paradigm specification* technique whereby different aspects of the system can be written in different (formal) notations. For example, the expected behaviour of the system can be specified using languages such as LOTOS, with QoS management functions specified using timed automata. We also provide support for the specification of real-time constraints using real-time temporal logic. Further details of this approach can be found in [Blair99a]. Using this approach, we can verify not only the QoS management subsystem itself, but also its interaction with the rest of the system. The approach to this is based on a *composition algorithm*, which relies on a common underlying semantic model based on *labelled transition systems* [Blair99a]. The overall approach is summarised in figure 2.

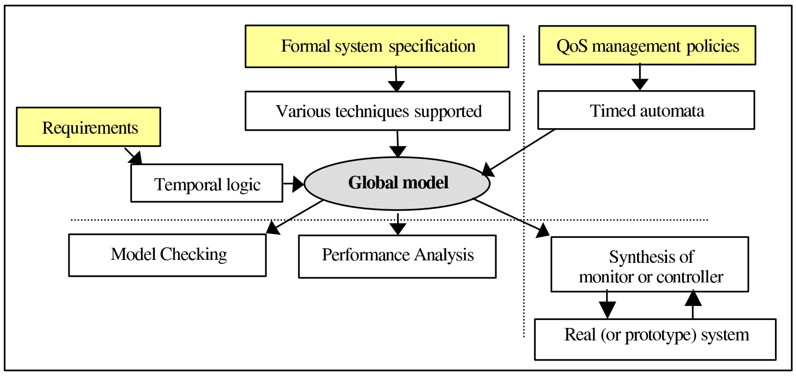

Figure 2: Our multi-paradigm specification approach.

We have also developed a *tool suite* to support this process. Currently, this supports the composition of specifications written in LOTOS, timed automata, and the real-time temporal logic QTL [Blair94]. The output of this process is a combined timed automaton, the behaviour of which can be investigated using either a simulator (to interactively step through behaviour) or a model checker (to verify the desired properties as expressed using temporal logic). In essence, the tool allows us to verify QoS management functions in isolation or in combination with other system behaviour.

Note that timed automata, once verified by the tool, can be used directly as scripts for QoS management functions. To support this, the tool outputs timed automata using a standardised format (FC2 [Madelaine93]). This format is also used as a scripting language for respective QoS management functions. Using this standardised format also encourages interoperability with other formal tools.

5. IMPLEMENTATION

5.1. The Underlying Platform

An implementation of the reflective middleware architecture and associated QoS management functions has been carried out using Python [Watters96]. Python was used for three main reasons. Firstly, the language already has some reflective facilities, on which it has proved to be relatively easy to build our meta-models. Secondly, the language is an ideal vehicle for rapid prototyping, due to its interpreted nature and flexible typing. Thirdly, the language also provides good support for distributed programming. The one drawback with using Python is that, due to its interpreted nature, performance is not comparable to a compiled language. This is however not viewed as critical at this stage of the research. We revisit this argument in section 7 below.

The platform implements the programming model introduced in section 3.1. In more detail, a `Component` class provides one or more interfaces, where an interface can be of type operational, stream or signal. Operational interfaces support operations in the traditional CORBA sense. In contrast, stream interfaces support either the production or consumption of continuous media data. Finally, signal interfaces emit or receive primitive, asynchronous events. Component interfaces in the same address space can be bound together using a `localBind()` primitive. To cross address spaces (or machines), it is necessary however to create a binding component (i.e. explicit binding as discussed in section 3.1).

The `Component` class also supports access to the different meta-models defined in our architecture. At present, the platform focuses mainly on structural reflection with complete implementations of the encapsulation and composition meta-models (although minimal implementations of the other meta-models also exist). The encapsulation meta-model provides operations to inspect the target interfaces (c.f Java introspection), to set, get or delete attributes, to add or delete methods, and to introduce or remove pre-and post-methods. The composition meta-model provides a method to inspect the internal structure of a composite component, returning a representation of the underlying component graph (see section 3.2.1). Operations are then provided to add or remove a component, to create or break a local binding, and to atomically replace one component with another. Access to the meta-spaces is provided by the operations `encapsulation()` and `composition()` respectively (defined on both components and interfaces).

The platform also provides an underlying engineering model heavily influenced by RM-ODP [Blair97]. In particular, the platform provides a `Capsule` class, representing a distinct address space. This class supports operations to create a new component within that address space or to register an existing component with that address space (thus making it accessible from that capsule). Capsules also provide the `localBind()` primitive mentioned above, as well as a corresponding `breakBinding()` operation. In addition, the platform supports a simple name server, together with an extensible set of factories to add new components to the environment (e.g. to create a new binding component). Factories are accessed through the capsule interface as described above.

Crucially, the platform supports the creation of management components for dynamic QoS management (as discussed in section 4 above). Such management components act as interpreters for timed automata (described using FC2), and interact with other objects using signal interfaces. We illustrate the use of the management components with a simple example of a QoS-managed audio stream, which has been implemented on the platform. A second, more detailed example can also be found in [Gancedo99], which also includes a more thorough evaluation of the approach.

5.2. A QoS Management Example

In the example, we have an audio source and sink (on different machines) connected by an audio stream binding component. As part of dynamic QoS management, we would like to monitor the state of the sink's buffer to ensure that it is not "full too often" (FTO) or "empty too often" (ETO). To achieve this, the sink monitor (*snk_monitor*) receives, as input, events *buf_full* and *buf_empty* from the sink's buffer. We assume that an appropriate event collector (which exploits the underlying meta-models to locate and monitor the relevant behaviour) generates these events. The monitor sets a period, P, in which the above events are counted (*nf* and *ne* respectively). If the number of full (resp. empty) events occur more than a pre-determined number of times (N) within period P, then the event *FTO!* (resp. *ETO!*) is output. At the end of the period, all

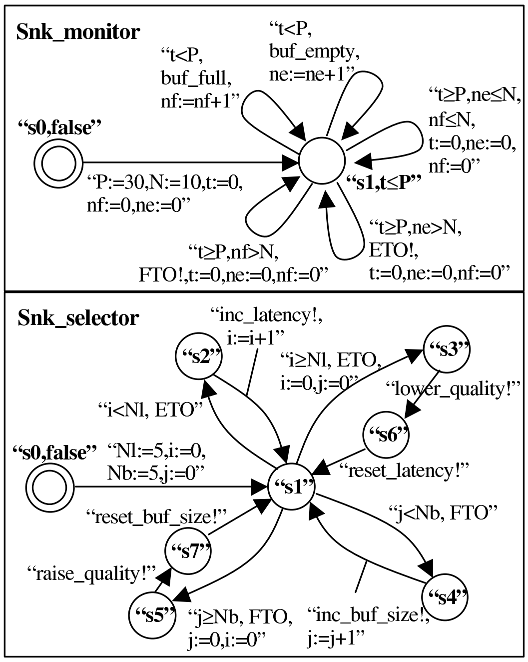

Figure 3: Automata to manage the sink's buffer.

variables are reset.

These two output events are then treated as input events by the sink strategy selector (*snk_selector*). There are various options that the strategy selector can take depending on the sequencing and frequency of the inputs. Firstly, we address the receipt of ETO events, i.e. the sink's buffer is empty too often. We note that the latency of the presentation can be increased to allow the buffer a fixed period of time to (partially) fill up with data (output event *inc_latency!*). However, since we cannot keep increasing the latency indefinitely, we count the number of latency adjustments (i) up to a limit (Nl). Once Nl adjustments have been made, the selector adopts the strategy of asking the source to lower its quality (by emitting an appropriate signal). This strategy also causes the latency to be reset and we try again. If, however, we address the receipt of FTO (full too often) events, the strategy is to adjust the buffer size. Again there is a limit to how many times this should be done. Hence, we count, as above, the number of buffer adjustments (j) up to a limit (Nb). This time, once Nb adjustments have been made, the selector signals to the source to try and raise its quality. This also resets the buffer size. The overall automata are shown in figure 3.

As above, we assume the existence of appropriate strategy activators to bridge between the QoS management subsystem and the underlying functional components. For example, the strategy activator responsible for increasing buffer size operates through the compositional meta-model to locate and re-configure the buffering component.

In this example, it is also possible to combine the monitor and strategy selector into one automaton, using the tool described above (e.g. to improve the efficiency of the QoS management subsystem). This produces a timed automaton with nine states.

Note that this example could also be extended by providing a meta-policy. For example, a meta-manager could monitor the behaviour of the management subsystem and detect the number of events generated by the controller (to change the latency associated with the buffer or the quality of audio sent by the source). If such events are generated too frequently, it is likely that the management subsystem is less than optimal (it could be oscillating unnecessarily between states). Action can then be taken to alter some of the key values associated with the monitor and strategy selector, e.g. N and Nb. A more radical strategy would be to change the strategy selector completely should the current strategy be seen to behave poorly. This is achieved by downloading a new automaton described in FC2.

6. RELATED WORK

6.1. Reflection in Distributed Systems

There is growing interest in the use of reflection in distributed systems. Pioneering work in this general area was carried out by McAffer [McAffer96]. With respect to middleware in particular, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) [Singhai97, Román99], focussing on reification of invocation marshalling and dispatching. In addition, researchers at APM have developed an experimental middleware platform called FlexiNet [Hayton97], which 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 computing [Manola93], i.e. a minimal object model which can be specialised through reflection. Finally, researchers at the Ecole des Mines de Nantes are also investigating the use of reflection in proxy mechanisms for ORBs [Ledoux97].

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 [Okamura92]. From this list, it can be seen that AL/1-D does however support a resources model. This resource model supports reification of scheduling and garbage collection of objects (but in a relatively limited way compared to our approach). Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R [Watanabe88] and CodA [McAffer96]. 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.

Our use of component graphs is inspired by researchers at JAIST in Japan [Hokimoto96]. In their system, adaptation is handled through the use of control scripts written in TCL. Although related to our proposals, the JAIST work does not provide access to the internal details of communication components. Furthermore, the work is not integrated into a middleware platform. The designers of the VuSystem [Lindblad96] and Mash [McCanne97] advocate similar approaches. The same criticisms however also apply to their designs. Microsoft's ActiveX software [Microsoft99] also uses component graphs. This software, however, does not address distribution of component graphs. In addition, the graph is not re-configurable during the presentation of a media stream.

6.2. QoS Management in Distributed Systems

There are many projects investigating QoS management in middleware platforms. BBN's QuO (QUality Objects) project [Vanegas98] is perhaps the closest in approach to our own work in that it adopts a similar open implementation philosophy. QuO, however, adopts an approach reminiscent of aspect-oriented programming [Kiczales97] to specify different aspects of QoS support using a range of specialised (high-level) languages. As such, it provides a more declarative, as opposed to procedural, approach to QoS management.

Similarly, researchers at the University of Illinois have developed a task control model to support dynamic reconfiguration in middleware platforms [Li99]. This project is complementary to the Illinois work on reflective middleware platforms mentioned above, exploiting the openness of the underlying platform for more fine-grained control. Their approach is based on the use of a fuzzy logic inference engine together with a rule base of possible adaptations, in contrast to our use of timed automata.

Other notable research projects include DIMMA [Donaldson98], Tao [Schmidt97], Jonathan [Dumant97], GOPI [Coulson99], and ERDoS [Chatterjee97]. In addition, the OMG have carried out important work in this area with their specifications for the "Control and Management of A/V Streams" [OMG97] and "Realtime CORBA" [OMG98]. However, these projects all tend either to offer only restricted access to underlying middleware functions, or to focus on more static aspects of QoS management (specification, configuration, and resource reservation).

Finally, there is a considerable body of research on specific techniques for QoS management in distributed systems (not necessarily related to middleware platforms). For example, many of the most influential techniques have emerged from Internet applications such as the Mbone tools, e.g. *vic*, [McCanne95]. Other important techniques include the use of filters [Yeadon96], and receiver-driven adaptation through layered encoding [McCanne96]. There is also a growing trend to provide toolkits incorporating such approaches to support the construction of adaptive multimedia applications, e.g. SWiFT [Goel98, Walpole97], Djinn [Mitchell98], and the JAIST research mentioned above [Hokimoto96].

7. CONCLUDING REMARKS

This paper has considered the role of reflection in supporting important QoS management functions. More specifically, the paper has described the design of a novel reflective middleware platform and examined how the dynamic QoS management functions of monitoring and adaptation can be supported in this design. The most important features of the approach are:

i) a procedural approach is adopted, whereby QoS management functions operate on a component graph of underlying components exposed via meta-space;
ii) QoS management is introduced via management components which communicate with other events through event notification (thus providing an important separation of concerns);
iii) three styles of management component are identified, namely monitors, strategy selectors and strategy activators;
iv) monitors and strategy selectors are typically written using a timed automata notation thus encouraging the formal verification of QoS management subsystems.

We believe that the use of reflection provides important benefits over more traditional approaches to QoS management. For example, reflection provides a principled approach to providing access to underlying components. This is crucial in supporting monitoring and adaptation aspects of QoS management. The approach is also dynamic in that management components can be added or removed at any time. In addition, reflection enables the introduction of sophisticated management structures involving both policies and meta-policies.

Finally, the experience from our pilot implementation work (see section 5) provides initial validation for our claims. Although performance is not a primary concern at this stage, the platform was able to deliver the required real-time performance for the audio stream example. In the future, we plan to carry out further implementation experiments and to extend the work by considering more sophisticated examples of QoS management policies and indeed meta-policies. Ongoing studies are also focussing on the issue of performance. In particular, we are currently implementing a lightweight, reflective component model, based on COM, offering access to each of the meta-space models. A first implementation of this component model has now been completed. We are now using the reflective components to construct a configurable and re-configurable CORBA-base platform following the architecture described above. The ultimate aim of this work is to demonstrate that flexibility is not necessarily at the expense of performance.

ACKNOWLEDGEMENTS

The research described in this paper is partly funded by CNET, France Telecom (CNET Grant 96-1B-239). Particular thanks are due to Jean-Bernard Stefani and his group at CNET for many useful discussions on reflection and distributed systems. The work on specification and formal verification of QoS properties is funded by EPSRC (Research Grant GR/L28890). We would also like to acknowledge the contributions of our partners on the CORBAng project (next generation CORBA) at UniK, and the Universities of Oslo and Tromsø (all in Norway). Particular thanks to Frank Eliassen, Vera Goebel, Oyvind Hanssen and Thomas Plagemann. Finally, we would like to acknowledge the contributions of a number of researchers at Lancaster to the ideas described in this paper, namely Mike Clarke, Fabio Costa, Tom Fitzpatrick, Hector Duran, Nikos Parlavantzas, Philippe Robin and Katia Saikoski.

REFERENCES

[Agha91] Agha, G., "The Structure and Semantics of Actor Languages", Lecture Notes in Computer Science, Vol. 489, pp 1-59, Springer, 1991.

[Blair94] Blair, L., "The Formal Specification and Verification of Distributed Multimedia Systems", PhD thesis, Available from the Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, 1994.

[Blair97] Blair, G.S., J.B. Stefani, "Open Distributed Processing and Multimedia", Addison-Wesley, 1997.

[Blair98] Blair, G.S., G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, 1998.

[Blair99a] Blair, L., G.S. Blair, "Composition in Multi-paradigm Specification Techniques", In Proceedings of the 3rd International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'99), February 15th-18th, Florence, Italy, P. Ciancarini, A. Fantechi, R. Gorrieri (eds), Kluwer, 1999.

[Blair99b] Blair, G.S., F. Costa, G. Coulson, H. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, J.B. Stefani, "The Design of a Resource-Aware Reflective Middleware Architecture", Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), St-Malo, France, Springer-Verlag, LNCS, Vol. 1616, pp115-134, 1999.

[Chatterjee97] S. Chatterjee, J. Sydir and B. Sabata, "Modeling Applications for Adaptive QoS-based Resource Management", Proceedings of the 2nd IEEE High Assurance Systems Engineering Workshop, pp 194-201, Bethesda, Maryland, August 1997.

[Costa98] Costa, F., G.S. Blair, G. Coulson, "Experiments with Reflective Middleware", Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, ECOOP'98 Workshop Reader, Springer-Verlag, 1998.

[Coulson99] Coulson, G., "A Configurable Multimedia Middleware Platform", IEEE Multimedia, Vol 6, pp 62-76, No 1, January - March 1999.

[Donaldson98] Donaldson, D., Faupel, M., Hayton, R., Herbert, A., Howarth, N., Kramer, A., MacMillan, I., Otway D. and Waterhouse, S., "DIMMA - A Multi-media ORB", Proc. Middleware '98, The Low Wood Hotel, Ambleside, England, September 1998.

[Dumant97] Dumant, B., F. Horn, F. Dang Tran, J.B. Stefani, "Jonathan: An Open Distributed Processing Environment in Java", Proc. IFIP International

Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, September 1998.

[Fitzpatrick98] Fitzpatrick, T., G.S. Blair, G. Coulson, N. Davies, P. Robin, "Supporting Adaptive Multimedia Applications through Open Bindings", Proceedings of the 4th International Conference on Configurable Distributed Systems, IEEE, 1998.

[Gancedo99] Gancedo, D.S., "QoSMonAuTA: QoS Monitoring and Adaptation using Timed Automata", MSc Thesis, Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK, September 1999.

[Goel98] Goel, A., D. Steere, C. Pu, and J. Walpole, "SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit", Technical Report 98-009, Computer Science and Engineering Dept., Oregon Graduate Institute of Science and Technology, USA, 1999.

[Hayton97] Hayton, R., "FlexiNet Open ORB Framework", APM Technical Report 2047.01.00, APM Ltd, Poseidon House, Castle Park, Cambridge, UK, 1997.

[Hokimoto96] Hokimoto, A, T. Nakajima, "An Approach for Constructing Mobile Applications using Service Proxies", Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96), IEEE, 1996.

[Kiczales91] Kiczales, G., J. des Rivières, D.G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, 1991.

[Kiczales97] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingtier, J. Irwin, "Aspect-Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Lecture Notes in Computer Science, Vol. 1241, Springer-Verlag, June 1997.

[Román99] Román, M., F. Kon, R. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case", Proceedings of the ICDCS'99 Workshop on Middleware, Austin, Texas, May-June 1999.

[Ledoux97] Ledoux, T., "Implementing Proxy Objects in a Reflective ORB", Proc. ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation, Jyväskylä, Finland, 1997.

[Li99] Li, B. and K. Nahrstedt, "Dynamic Reconfiguration for Complex Multimedia Applications,Proceedings of the IEEE International Conference on Multimedia Computing and Systems (IEEE Multimedia Systems '99), Florence, Italy, June 7-11, 1999.

[Lindblad96] Lindblad, C.J., D.L. Tennenhouse, "The VuSystem: A Programming System for Computer-Intensive Multimedia", Journal of Selected Areas in Communications, Vol. 14, No. 7, pp 1298-1313, IEEE, 1996.

[Madelaine93] E. Madelaine, R. de Simone, "FC2: Reference Manual Version 1.1", see http://www.inria.fr/meije/verification/doc.html, 1994.

[Manola93] Manola, F., "MetaObject Protocol Concepts for a "RISC" Object Model", Technical Report TR-0244-12-93-165, GTE Laboratories, 40 Sylvan Road, Waltham, MA 02254, USA, 1993.

[McAffer96] McAffer, J., "Meta-Level Architecture Support for Distributed Objects", In Proceedings of Reflection 96, G. Kiczales (end), pp 39-62, San Francisco; Also available from Department of Information Science, The University of Tokyo, 1996.

[McCanne95] McCanne, S. and Jacobson, V., "vic: A Flexible Framework for Packet Video, Proc. ACM Multimedia '95, San Francisco, USA, pp 511-522, November 1995.

[McCanne96] McCanne, S. and Jacobson, V., "Receiver Driven Layered Multicast", Proc. ACM SIGCOMM'96, pp 117-130, Stanford, CA, USA, 1996.

[McCanne97] McCanne, S., 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, D. Wu, "Towards a Common Infrastructure for Multimedia-Networking Middleware", Proc. 7th International Conference on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97), St Louis, Missouri, 1997.

[Microsoft99] Microsoft ActiveX Home Page: http://www.microsoft.com/com/tech/activex.asp.

[Mitchell98] Mitchell, S., H. Naguib, G. Coulouris, and T. Kindberg, "A QoS Support Framework for Dynamically Reconfigurable Multimedia Applications", Proc. DAIS'99, Helsinki, Finland, June 1999.

[Okamura92] Okamura, H., Y. Ishikawa, M. Tokoro, "AL-1/d: A Distributed Programming System with

Multi-Model Reflection Framework", Proceedings of the Workshop on New Models for Software Architecture, November 1992.

[OMG97] Control and Management of Audio Visual Streams, OMG Document number telecom/97-05-07; available from http://www.omg.org/library/schedule/AV_Streams_RTF.htm

[OMG98] Realtime CORBA V1.1, Initial Submission to Realtime SIG's RFP on Realtime CORBA, OMG Document number orbos/98-01-08.

[OMG99] The Common Object Request Broker: Architecture and Specification versions 3.0., available at http://www.omg.org/.

[Saikoski99] Saikoski, K.B. and Coulson, G., "Adaptive Groups in OpenORB", Proceedings of the 6th Doctoral Consortium on Advanced Information System Engineering (CAiSE'99 DC), Heidelberg, Germany, 14-16 June 1999.

[Schmidt97] Schmidt, D.C., R. Bector, D. Levine, S. Mungee, G. Parulkar, "An ORB End System Architecture for Statically Scheduled Real-Time Applications", Proceedings of the Workshop on Middleware for Real-Time Systems and Services", IEEE, San Francisco, 1997.

[Singhai97] Singhai, A., R. Campbell, "Reflective ORBs: Supporting Robust, Time-critical Distribution", Proc. ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems, Jyväskylä, Finland, 1997.

[Smith82] Smith, B.C., "Procedural Reflection in Programming Languages", PhD Thesis, MIT, Available as MIT Laboratory of Computer Science Technical Report 272, Cambridge, Mass., 1982.

[Szyperski98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.

[Vanegas98] Vanegas, R., J. Zinky, J. Loyall, D. Karr, R. Schantz, D. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, September 1998.

[Walpole97] Walpole, J., R. Koster, S. Cen, C. Cowan, D. Maier, D. McNamee, C. Pu, D. Steere, and L. Yu, "A Player for Adaptive MPEG Video Streaming Over The Internet", Proc. 26th AIPR-97, SPIE, Washington DC, October 15-17, 1997.

[Watanabe88] Watanabe, T., A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language", In Proceedings of OOPSLA'88, Vol. 23 of ACM SIGPLAN Notices, pp 306-315, ACM Press, 1988; Also available as Chapter 3 of "Object-Oriented Concurrent Programming", A. Yonezawa, M. Tokoro (eds.), pp 45-70, MIT Press, 1987.

[Watters96] Watters, A., G. van Rossum, J. Ahlstrom, "Internet Programming with Python", Henry Holt (MIS/M&T Books), 1996.

[Yeadon96] Yeadon, N., Garcia, F., Shepherd, D., and Hutchison, D., "Filters: QoS Support Mechanisms for Multipeer Communications, IEEE Journal on Selected Areas in Communications, Special Issue on Distributed Multimedia Systems and Technology, Vol. 14, No. 7, pp 1245-1262, September 1996.

[Yokote92] Yokote, Y., "The Apertos Reflective Operating System: The Concept and Its Implementation", In Proceedings of OOPSLA'92, ACM SIGPLAN Notices, Vol. 28, pp 414-434, ACM Press, 1992.

Notes

1. This journal is available online at http://ioj.iee.org.uk/journals/ip-sen/. Volume 147, Issue 1 is available at http://ioj.iee.org.uk/journals/ip-sen/2000/01/. More information is available from http://www.iee.org.uk/publish/journals/