QoS Management Specification Support for Multimedia Middleware

Hector A. Duran-Limon, Gordon S. Blair
Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK.
Tel: +44 1524 593141
Fax: +44 1524 593608
{duranlim, gordon}@comp.lancs.ac.uk

Abstract: Middleware technologies are now widely used in order to provide support for the interaction of systems relying on different hardware and operating systems. At present middleware platforms, however, do not provide enough support for both the configuration and reconfiguration of quality of service (QoS) management aspects of real-time applications such as distributed multimedia systems. That is, current middleware only provides support for the low-level specification of QoS properties. This paper presents an architecture description language (ADL) called Xelha for the high-level specification of QoS management in multimedia middleware whereas lower-level aspects can be tuned by using an aspect-oriented suite of languages referred to as resource configuration description language (RCDL). Tool support is also provided for the interpretation of the Xelha and RCDL languages.

1. Introduction

Middleware has recently been applied to a wider range of application domains including distributed database, safety-critical and real-time systems. Some of these domains are time-critical and demand configurable middleware in order to meet the requirements of the target platform. For instance, distributed multimedia applications have specific quality of service (QoS) properties which translate into particular requirements in the amount of resources that are needed to work properly. These resources include bandwidth, memory buffer and CPU. However, there is currently a lack of support for the configuration of QoS management in middleware. Some attempts have been made to introduce resource management configuration capabilities in middleware. Nevertheless, such solutions usually support the configuration of quality of service (QoS) properties by employing low-level specifications. As a result, considerable complexity is introduced in the system development process which is also error-prone. For example, Real-time CORBA provides facilities for the specification of thread pools and memory buffers, but no support is provided for the definition of higher-level aspects such as the end-to-end delay of interactions and delay variations (i.e. jitter). Moreover, some solutions, such as CORBA, do not provide any support for the specification of the mechanisms to attain the desired level of QoS. Mechanisms for QoS reconfiguration are needed to gracefully adapt the systems when unexpected changes are introduced into the environment. In addition, some of the approaches usually model resource management for single client-object interactions which does not represent any problem for small-scale applications. However, for larger-scale applications the code complexity increases considerably.

This paper presents language support for the specification of QoS management in multimedia middleware. In particular, an ADL \(^1\) called Xelha, and RCDL, a low-level resource description language are introduced. The former is concerned with the high-level design of QoS management issues whereas

\(^1\) ADLs represent formal notations for describing software architectures in terms of coarse-grained components and connectors. Further information about ADLs can be found in (Shaw and Garlan, 1996; Medvidovic and Taylor, 2000).
the latter is a suite of aspect-oriented languages\(^2\) that provide support for the low-level specification of the system resources. In addition, tool support is provided to map Xelha specifications to RCDL descriptions.

The languages are derived from a resource framework which is part of OpenORB (Blair et al., 2001), a reflective middleware architecture. The resource framework includes both a resource model and a task model. The resource model offers various levels of abstraction for the representation of the resources subsystem whereas the latter provides the means for the high-level analysis and design of the system’s resource management. The task model also allows for the resource management of both fine- and coarse grained interactions.

The paper is structured as follows. Section 2 introduces our middleware architecture and the resource framework. Section 3 then presents the Xelha ADL. The RCDL is introduced in section 4. Following this, section 5 describes the implementation of a tool for the interpretation of both Xelha and RCDL. An evaluation of the approach is presented in section 6 and section 7 comments on related work. Finally, some concluding remarks are included in section 8.

2. Middleware Architecture

2.1 Overall Approach

The OpenORB framework is basically a componentised reflective middleware. Reflection allows for opening up the middleware in a principled way. Complementary to this, the component technology introduces more configuration and reconfiguration capabilities into distributed applications and increases the level of reuse. Within the context of OpenORB a component is basically “a unit of composition with contractually specified interfaces and explicit context dependencies only” (Szymerski, 1998). The component model used in the OpenORB architecture is greatly influenced by the Computational Model from RM-ODP (Blair and Stefani, 1997). Hence, components within this model have several interfaces. Not only operational interfaces but also stream and signal interfaces are supported. Moreover, explicit bindings are supported which offer more control over the communication path between object interfaces.

The meta-space of the platform is structured, but not restricted, as a set of four orthogonal meta-models (interface, architecture, interception and resources.) The interface meta-model allows for the inspection of all interfaces offered by a given component. Importantly, interfaces are immutable. That is, it is not allowed to change the internal implementation of an interface. The architecture meta-model provides a representation of the component connexions of a composite component. The interception meta-model is then in charge of introducing additional behaviour to a component interface. This meta-model is realised as dynamic plugable interceptors, which enable the insertion of pre- and post-behaviour. Finally, the resource meta-model is concerned with both the resource awareness and resource management of objects in the platform. A more comprehensive description of the resource meta-model is presented below. Further details of the overall architecture can be found in the literature

\(^2\) Aspect-oriented programming (Kiczales et al., 1997) allows for the decomposition of a program into aspects that cross-cut each other.
(Blair et al., 2001; Coulson et al., 2001; Costa et al., 2000; Costa and Blair, 2000; Saikoski et al., 2000; Parlavantzas et al., 2000; Moreira et al., 2001; Blair et al., 2000a; Blair and Campbell, 2000; Costa, 2001). For example, detailed descriptions of the four meta-models can be found in (Blair et al., 2001).

2.2 The Resource Model

The most important elements of the resource model are abstract resources, resource factories and resource managers (ReTINA, 1999; Blair et al., 1999a; Duran-Limon and Blair, 2000; Duran-Limon and Blair, 2002). Abstract resources explicitly represent system resources. In addition, there may be various levels of abstraction in which higher-level resources are constructed on top of lower-level resources. Resource managers are responsible for managing resources, that is, such managers either map or multiplex higher-level resources on top of lower-level resources. Furthermore, resource schedulers are a specialisation of managers and are in charge of managing processing resources such as threads or virtual processors (or kernel threads). Lastly, the main duty of resource factories is to create abstract resources. For this purpose, higher-level factories make use of lower-level factories to construct higher-level resources. The resource model then consists of three complementary hierarchies corresponding to the main elements of the resource model. Importantly, virtual task machines (VTMs) are top-level resource abstractions and they may encompass several kinds of resources (e.g. CPU, memory and network resources) allocated to a particular task. Further details regarding the resource model can be found in (Duran-Limon and Blair, 2000; Duran-Limon and Blair, 2002).

2.2.2 The Task Model

There is a one-to-one mapping between tasks and VTM s within an address space. Hence, a VTM represents a virtual machine in charge of supporting the execution of its associated task. A task is defined as a logic unit of computation which has an amount of resources allocated. Examples of tasks are activities performed by the system such as transmitting audio over the network or compressing a video image. From the programmatic point of view, a task may involve either a single invocation sequence or multiple invocation sequences. The simplest case for a sequence is that whereby only one operation is invoked. A task may span the boundaries of an object and even those of an address space. Composite tasks include two or more sub-tasks. Sub-tasks that are not further partitioned are called primitive tasks and are only related to a single address space. Importantly, different tasks may be interconnected. For instance, an object running one task may invoke another object concerned with a different task. Such a method invocation represents a task switching point. Thus, a task switching point corresponds to a change in the underlying resource pool to support the execution of the task that has come into play. A detailed description of the task model can be found in (Duran-Limon, 2001).

2.2.3 Requirements for Language Support

As mentioned previously, the authors believe that multimedia middleware platforms should be both configurable and dynamically reconfigurable. That is, time sensitive applications, such as multimedia systems, demand certain QoS assurances. As a consequence, language support is required for specifying the QoS management configuration of middleware in order to meet the needs of both the
application domain and the platform of deployment. In addition, multimedia systems are highly sensitive to changes introduced into their environment. Therefore, language support is required for specifying the QoS management reconfiguration of middleware to sustain the desired level of QoS when unanticipated changes occur.

A set of requirements is identified for a language aiming to support both configuration and reconfiguration of QoS management in multimedia middleware.

- **Support for coarse- and fine-grained specifications.** The language should provide semantics for the coarse-grained specification of the middleware system. Examples of coarse-grained specifications include the definition of the services configured as coarse block units without revealing implementation details. This approach aims at diminishing the overall complexity of the system configuration. Hence, the user can focus on the high-level configuration design of the system. In addition, the language should offer support for the description of fine-grained details. The aim of the lower-level description language is to map coarse-grained descriptions to fine-grained details of the platform of deployment. That is, the coarse-grained description language is concerned with the high-level analysis and design of the system whereas the fine-grained description language is closely related to the implementation details of the system.

- **Separation of concerns.** The language should ideally provide facilities for the separation of concerns. By doing so, the programmer is able to tackle different issues in a clearer manner. The result of this approach is that the overall complexity of the system development is diminished.

- **Expressiveness.** The description language should have enough power of expressiveness for specifying both configuration and reconfiguration QoS management aspects for distributed multimedia systems.

- **Tool support.** The description language should have associated tool support. The purpose of this is to provide facilities for the design and analysis of a system. Importantly, the tool(s) should also provide support for code system generation. Such generated code may be either lower-level code or executable programs.

In the following sections the Xelha and RCDL description languages are introduced for the support of QoS management specification in multimedia middleware.

### 3. Xelha: a Resource-Aware ADL

#### 3.1 Components, Connectors and Interfaces
As with most ADLs, the language is based on components and connectors. Components represent elements possibly residing in different capsules whereas connectors model component interactions. Both components and connectors may include other composite and non-composite elements. Hence, a distributed system is represented as a hierarchical composition in terms of both composite components...
and composite connectors. Such a hierarchical composition is similar to that offered by Darwin (Magee et al., 1995) for the construction of composite components. Hence, the granularity of a component type may range from a primitive component type to a complex system configuration type. The top-level component, which is at the top of the composition hierarchy, models the whole system architecture.

![Xelha Specification](image)

**Figure 1.** Top-level Component Specification

The language is specified in extended BNF (ISO/IEC, 1996) and the complete specification is included in Appendix A. The structure of the top-level component is shown in figure 1. The components section defines the constituents of a composite component which encompasses two or more components. This is achieved by defining both the component type and the capsule where the component is to be created as defined in figure 2.

![Figure 2. Definition of some of the Internal Elements of the Top-level Component](image)

An example of how an internal component is defined is as follows:

```plaintext
components:
  srcStub: SrcStub, srcCapsule
```

```plaintext
interfaces:
  IN: SrcStubIN, ( srcStub, IN )
```

The component name srcStub is associated with the component type SrcStub and the capsule srcCapsule.

Similarly, the connectors section specifies the internal connectors of a composite component. The external interfaces of the component are then defined in the interfaces section. This specification also defines the interface name IN. The interface name is then a reference to the interface of the same name belonging to the component srcStub whereby the defined interface type is SrcStubIN.

In addition, the composition graph section defines how these components are interconnected. For this purpose, in the interface section the involved component interfaces are associated with unique names. Following this, interface attachments are defined in the edges section. As an example consider the following composition graph:

```plaintext
composition graph:
  interfaces:
    srcOUT: ( srcStub, OUT )
```
The interfaces section defines the interface name srcOUT for the interface of name OUT that belongs to the component named srcStub. Similarly, the interface name streamConnIn is defined as the interface IN of the component streamConn. The edges section defines then the interfaces to be bound.

Finally, both the tasks and services sections allow us to specify the QoS management properties associated with a component. A detailed explanation of both task and service specifications is introduced in the next section. It should be noted that specifications of primitive components only include the interfaces and tasks sections whereas composite components are concerned with all of them except the service section. Only the top-level component covers the service section as discussed below.

In addition, connector specifications are similar to component definitions as shown in figure 3. Both of them allow for inheritance. Hence, when sub-typing a component type, the whole structure of it is inherited (e.g. internal components, interfaces, component graph, etc). Note that connector specifications additionally associate a connector style with the connector definition. There are three connector styles: Operational connections, signal connections and stream connections. Furthermore, connector specifications separate those interfaces that are part of the interaction protocol from the interfaces that provide some control over the connector. The purpose of this is to perform some type-checking to ensure that the style of the interfaces involved in the interaction protocol conform with the style of the connector (see below). In case of distributed connectors, the participating capsules are passed as parameters to the connector type.

![Figure 3. Specification of Component and Connector](image)

Interface types are specified outside the definition of a component as shown in figure 4. Interfaces are associated with interaction styles which conform to the connection styles described above. Furthermore, interfaces are defined in terms of both the operations they provide and the operations they require. An interface may also be extended whereby the extended interface inherits all provided and required operation definitions that are specified by the base interface.
Figure 4. Interface Specification

The language also supports the specification of dynamic architectures, i.e. architectures that may experience changes at run-time. This feature allows us to specify a range of architectural types, defined within a type hierarchy, that can be instantiated as a result of accessing a service provided by a component. Thus, component and connector definitions are considered to be type definitions similar to that of a class. Component types may also be generic and, as a consequence, parameterised. The purpose of this is to introduce the ability of defining an architecture of a composite component in which each valid internal component type may be part of a sub-type hierarchy. Such an approach allows us to reuse component configuration definitions. We introduce the optional construct inst within the definition of an operation to denote that the invocation of such an operation will result in the dynamic instantiation of the component type (or any of its sub-types). For instance, consider a stream connection server in charge of instantiating stream bindings as depicted in figure 5. The component userInterface is connected to the interface IN of the connections server. The specification of this interface is as follows:

```
Def Interface <operational> ConnServerInterf:
  provides:
    boolean new_connection(in string srcInterf,
                            in string sinkInterf,
                            in QoSspec qos):
      inst StreamConnection
```

The interface of the server exposes the operation new_connection() whose parameters are the interfaces to be bound and the required level of QoS for the binding. The invocation of the operation then results in the dynamic instantiation of the StreamConnection component type which is specified as follows:

```
Def component StreamConnection(StreamBinding bindingType,
                                 string srcCapsule, string sinkCapsule,
                                 SrcOUT srcInterf, SinkIN sinkInterf):
  connectors:
    binding: bindingType(srcCapsule,sinkCapsule)
  composition graph:
    interfaces:
      bindingIN: (binding, IN)
      bindingOUT: (binding, OUT)
    edges:
```
This is a parameterised component whose internal configuration includes two interfaces bound by the StreamBinding connector type. The inheritance hierarchy of this connector type is shown in figure 6. The hierarchy contains connectors for the transmission of both audio and video. In both cases, two different versions associated with a distinct level of QoS are offered. Therefore, diverse architectural components may be dynamically created as a result of invoking the operation new_connection().

![Figure 5. Dynamic System Architecture](image)

Finally, as an optional feature, the language allows components residing in the same address space to interact directly without the use of a connector. Thus, when component interactions are rather simple, such interactions may be modelled implicitly.

![Figure 6. The UML Inheritance Diagram of the StreamBinding Connector Type.](image)

### 3.2 QoS Management

#### 3.2.1 Overview

The ADL offers support for the specification of both the QoS properties provided by a service and the actions taken in order to attain the contracted level of QoS. The former corresponds to static QoS management aspects, which include the initial component configuration and the amount of resources required to provide the stated level of QoS (Blair et al., 2000b). The latter, in contrast, relates to dynamic QoS management aspects which in turn involves run-time monitoring and dynamic reconfiguration of both the component configuration and the configuration of resources (Blair et al.,
2000b). The support that Xelha provides for both static and dynamic QoS aspects are introduced in turn below.

### 3.2.2 Static QoS Management

Within Xelha, static QoS management aspects are specified in terms of task graphs together with their associated QoS specifications. There is a one-to-one relationship between a task graph and a statement of QoS properties.

![Figure 7. Example of a Component Configuration Associated with Task Graphs](image)

In our model, components are represented as rounded squares whereas connectors are depicted as rounded rectangles. In addition, similar to the work defined in (Rastofer and Bellosa, 2001) we employ the use case maps (UCM) notation (Buhr and Casselman, 1995; Buhr, 1998; Amyot, 1999) for the graphical representation of task graphs. The UCM notation includes elements for representing paths, start points and end points. Paths are depicted as lines that go across components. Start points are then represented as filled circles whereas end points are depicted as bars. Figure 7 shows an example of a component configuration associated with a task graph. The task path of the task transmitAu starts at the interface IN of the component AudioConnector_V1 and terminates at its interface OUT. The task graph goes across the components srcStub, streamConn and sinkStub. In addition, there are two subtasks defined, namely transmitAu.marshall and transmitAu.unmarshall. Each one of these task graphs has associated QoS properties. For instance, the end-to-end delay of task transmitAu is 20 ms.

![Figure 8. Representation of task switching points in UCM](image)

More complex interactions involving concurrent tasks may be represented by UCM’s ORs/ANDs notation. Hence, OR-forks are used to depict tasks emerging from a single path segment as shown in figure 8 (a). Moreover, OR-joins are used to indicate tasks that share the same path segment as depicted in figure 7 (b).
Importantly, single task switching points are represented by AND-joins, which indicate that one or more task graphs are switched into a single task graph, as shown in figure 8 (a). In addition, multiple task switching points are represented by a generic version of AND-fork/joins whereby multiple tasks graphs are switched into two or more task graphs, as shown in figure 8 (b). The obtained tasks from this task point are then represented by an OR-join since they all share the same task path segment.

<table>
<thead>
<tr>
<th>task = composite task</th>
<th>primitive task</th>
</tr>
</thead>
<tbody>
<tr>
<td>composite task =</td>
<td>“Def task”, task name, “includes”, 2 * task name, {task name}, “:”,</td>
</tr>
<tr>
<td></td>
<td>[“importance:”, digit],</td>
</tr>
<tr>
<td></td>
<td>“qos specifications:”, qos specifications,</td>
</tr>
<tr>
<td></td>
<td>[“qos management structure:”, qos mgmt. structure]];</td>
</tr>
<tr>
<td>primitive task =</td>
<td>“Def task”, task name, “:”,</td>
</tr>
<tr>
<td></td>
<td>[“importance:”, digit],</td>
</tr>
<tr>
<td></td>
<td>“switching points:”, end point, [“if “, task name],</td>
</tr>
<tr>
<td></td>
<td>“capsule:”, capsule name,</td>
</tr>
<tr>
<td></td>
<td>“qos specifications:”, qos specifications,</td>
</tr>
<tr>
<td></td>
<td>[“qos management structure:”, qos mgmt. structure]];</td>
</tr>
<tr>
<td>qos specifications =</td>
<td>{delay}, {jitter}, {throughput}, {packet loss}, {other}</td>
</tr>
<tr>
<td>delay =</td>
<td>“delay(“”, end point,”””, end point,”” ) = “”, digit,</td>
</tr>
<tr>
<td>jitter =</td>
<td>“jitter(“”, end point,”””, end point,”” ) = “”, digit,</td>
</tr>
<tr>
<td>throughput =</td>
<td>“throughput(“”, end point,”” ) = “”, digit,</td>
</tr>
<tr>
<td>packet loss =</td>
<td>“packet_loss(“”, end point,”””, end point,”” ) = “”, digit;</td>
</tr>
<tr>
<td>end point =</td>
<td>component name</td>
</tr>
</tbody>
</table>

**Figure 9. Task Specification**

Tasks may be either composite or primitive as shown in figure 9. In the first case, a task is composed of two or more sub-tasks which may in turn be composite or primitive. Sub-tasks that are not further partition are primitive. Interestingly, primitive task definitions include the specification of task switching points. Such points denote the operation that triggers the task switch and are expressed in conjunction with the associated interfaces and the components. This approach is sufficient to specify where tasks start and where they finish. “If” statements are optionally defined to determine whether a task switch should be performed according to the current task. Composite tasks are specified by defining the inclusion of other tasks as sub-tasks. In addition, the importance metric is used to define the criticality of a task; tasks with a high importance value have precedence over lower important tasks in case of resource contention. In addition, primitive tasks are associated with a particular capsule (i.e. address space).
Figure 10. Example of a Task Specification

QoS specifications allow us to define the QoS properties associated with a task graph. These properties are expressed in terms of three main QoS categories for distributed multimedia (Blair and Stefani, 1997). Firstly, timeliness properties are concerned with the end-to-end delay of multimedia interactions and delay variations (i.e. jitter), with both measured in milliseconds. Secondly, volume properties deal with the throughput of data and are measured either in frames delivered per second or in bytes per second. Finally, reliability properties refer to the permitted percentage of loss of media frames and bit error rates. As depicted in figure 10, QoS specifications are defined in terms of the task end points for each QoS property. Thus, a task end point is defined as a triplet including a component, an interface and an operation. It is worth mentioning that the QoS specifications in Xelha are not restricted to the set of QoS categories presented above and can be extended to cover other areas. An example of a task specification is presented in figure 10.

Figure 11. Specification of Services.

Finally, the services provided by a system architecture are defined within the section of services. This section applies only to top-level components and allows several types of services to be defined. A service type is basically a set of services with related semantics whereby each service offers a different level of QoS. Examples of service types are an audio communication service and a video conference service. A service type may then encompass more than one service as depicted in figure 11. The user-level of QoS offered by each service is specified. The tasks involved in a service are also defined. Importantly, reservation of resources is defined within the specification of the top-level composite component. Resources are reserved according to the specifications defined in the tasks sub-section in terms of the task name and the maximum number of instances for the task (as shown in figure 11). For
this purpose, the specifications can be processed by an interpreter whereby the QoS specifications of the task are translated into specific resources (e.g. amount of memory and percentage of CPU). This may be achieved by taking into account both the maximum desired number of task instances and platform specific resource requirements of the components involved. Such resource requirements can be obtained, for example, by measuring the component resource demands in a series of tests.

| qos mgt structure = “collector”: “interface name”; “interface type name: “component name; “interface name; “timed automaton”: “automaton name: “component type name; “strategy activator”: “activator name: “component type name; “QoS management graph”: “composition graph; |

**Figure 12. QoS Management Structure Specification.**

### 3.2.3 Dynamic QoS Management

Dynamic QoS management aspects are tackled through the use of a QoS management structure consisting of management components. A task may or may not have an associated QoS management structure. That is, when a task hierarchy is defined, a QoS management structure associated with the top-level task might be enough to determine the adaptation process for the whole hierarchy. However, finer-level of QoS management can be introduced by defining such adaptation structures for finer-level tasks.

Management components provide support for monitoring and control. Monitors and strategy selectors are specified by *timed automata* (Rajeev and Dill, 1994). Timed automata (TA) are finite state-transition graphs which also include timing (and data) constraints. The main feature of TA is that they can be simulated and formally verified for correctness and reachability. In addition, several graphs can be modelled and simulated separately, and afterwards combined to a single graph (Blair et al., 1999b; Blair and Blair, 2000). Xelha defines QoS management structures as shown in figure 12. Hence, monitors and strategy selectors are TA that are modelled separately and joined together afterwards in a single automaton whose behaviour is implemented by a pre-built component which can be automatically generated by the use of an associated tool (see section 5).

Event collectors are then represented as component operations which are responsible for observing events that occur in the component. All these operations are grouped in a single interface of the component. Finally, the strategy activator is concerned with another pre-built component that is in charge of carrying out the adaptation strategies.

The component configuration of the QoS management structure is specified in the *QoS management graph* section by defining how the management components are interconnected. As an example consider the connector type *AudioConnector* as depicted in figure 13. The internal structure of this connector includes a management structure whereby the automaton represents both the monitor and the strategy selector. The event collector is then represented by the interface *COLLECT* of the component *sinkStub*. In addition, the activator is able to receive signals from the automaton and can manipulate the stream connector by accessing its control interface. Such a QoS management
structure is specified in Xelha as shown in figure 12. The collector is specified by indicating the interface type SinkStub_Collect followed by the component and the interface name to be monitored.

The sections timed automaton and strategy activator define the component types that implement the automaton and the activator respectively. Finally, the composition graph section defines how the automaton and the activator are connected.

![Figure 13. Example of Connector with a QoS Management Structure.](image)

4. RCDL: a Set of Aspect Languages for the Specification of Resources

4.1 Overview

The RCDL is a lower-level description language which expands Xelha specifications into finer-grained aspects, including implementation details and platform specific concerns, as shown in table 1. Hence, Xelha definitions are mapped down into different aspects, thus, facilitating the analysis and further customisation of RCDL descriptions. More specifically, specification of the services provided by a system architecture are mapped down to service description language (SDL) descriptions which provide information about the QoS level, the task and the object class that are associated with a service. Task graph definitions in Xelha are translated to both task switch description language (TSDL) descriptions and task description language (TDL) descriptions. The former are concerned with the definitions of task switching points whereas the latter provides details about the resources assigned to tasks. Task QoS properties are further specialised by resource description language (RDL) descriptions, which define the specific resource requirements of a task instance according the characteristics of the platform of deployment. Finally, the QoS management structure is then described in the QoS management graph description language (QMGDL). Such graphs are defined on a per service type basis.

<table>
<thead>
<tr>
<th>RCDL’s Aspect Languages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Description Language (SDL)</td>
<td>Describes services in terms of the supported level of QoS, its associated task and the object class that implements the service.</td>
</tr>
<tr>
<td>Task Switch Description Language (TSDL)</td>
<td>Describes the object operations that trigger a task switch.</td>
</tr>
<tr>
<td>Task Description Language (TDL)</td>
<td>Describes the resources that are associated with a task.</td>
</tr>
<tr>
<td>Resource Description Language (RDL)</td>
<td>Describes the resources associated with a task instance.</td>
</tr>
<tr>
<td>QoS Management Graph Description Language (QMGDL)</td>
<td>Describes how management objects are interconnected.</td>
</tr>
</tbody>
</table>

Table 1. Aspect Languages of RCDL.
A summary of the RCDL’s aspect languages is presented in table 1 and more comprehensive description is presented below. Again, the language is specified with extended BNF and the complete specification is included in Appendix B.

4.2 Service Description Language

Application and middleware services are defined in SDL, also referred to as the service template, as shown in figure 14. Xelha specifications of service types are mapped to SDL descriptions. Within a service type, several services may be configured with each one having an associated QoS region, the task that supplies the required computational resources to provide the given level of QoS, and the (composite) object class that implements the service. The QoS region refers to the range of QoS values delineated by the minimum and maximum QoS levels offered by a service.

At run-time, service types are represented as operations in a services library. The join points for this aspect language are the invocation of such operations. Hence, the activation of a join point results in the selection of the most suitable service according to the requested level of QoS.

```
Service description = "Service Template:";
    {service type};;

service type = "Def service type", service type name, ";" {service};;

service = "Def service", service name, ";";
    "user_qos": user qos,
    "qos_region":
        "delay": minimum, ";", maximum,
        "throughput": minimum, ";", maximum,
        "jitter": minimum, ";", maximum,
        "packet_loss": minimum, ";", maximum,
    "task": task name,
    "object": object class name;

minimum = digit
maximum = digit
```

Figure 14. Specification of SDL

As an example of a service type consider the definition of the service type AudioCommService as depicted in figure 15. The service AudioComm_V1 is defined within this service type. The user-level of QoS for the service is defined. The specification of the QoS region then includes the minimum and maximum values for the main multimedia QoS categories. The service AudioComm_V1 then offers an end-to-end delay between 100 and 120 ms. The minimum supported throughput is 32 Kbits/seg whereas the maximum throughput offered is 64 Kbits/seg. The service allows a jitter between −10 and +10 ms. Lastly, the minimum and maximum percentage of packet loss supported by the service is 5% and 8% respectively. The specification of the service defines then the task transmitAudio_V1 is associated with the service. Finally, the service is implemented by the class AudioConnector_V1. It is worth mentioning that the QoS annotations defined in this section are not restricted to these QoS categories and may be extended to other domains.
4.3 Task Switch Description Language

The TSDL, also referred to as the object template, maps object operations to tasks as shown in figure 16. For this purpose, exported operations defined in Xelha are mapped to exported methods in TSDL. Similarly, imported methods in TSDL are derived from Xelha’s definitions. In addition, when two object interfaces are bound, imported methods must match the name and parameters of their related exported methods. Importantly, only exported operations may have associated tasks. Hence, task switches may only be performed by server objects. The definition of task switching points offers enough expressiveness for defining the interconnection of different tasks.

In addition, exported methods may be associated with multiple task switching points. To achieve this, an exported method is associated with a list of tasks. A task is selected from this list according to the current running task. For that purpose, “if” statements are introduced within the definition of an exported method in an object template as shown in figure 16. For instance, task f is selected if task g is the current running task that invokes the method z. The join points for this aspect language concern the invocation of object methods that are task switching points. The activation of a join point results in the selection of the appropriate task to be run.

![Figure 15. Example of SDL Descriptions](image)

![Figure 16. Example of TSDL Descriptions](image)
although they are associated with \( t_i \) and \( t_k \) respectively. Thus, the compressor will switch to task \( t_i \) if the filter was executed as part of task \( t_i \), otherwise it will switch to task \( t_k \), as shown in figure 17. Task switching may be achieved by defining “if” statements. In the second case, each of the methods of an object may be associated with a different task. For instance, consider a stub object with the marshal and unmarshal methods. These methods may be associated with different tasks since they are concerned with different activities, namely sending messages and receiving messages.

**Figure 17. Example of Object Configuration Associated with Two Tasks**

### 4.4 Task Description Language

The specification of the tasks associated with a service is defined by the TDL, also called the task template. Each of these tasks is related to a VTM in the task template. There is a different TDL specification for each address space in which the associated VTMs are defined. It should be noted that the scheduling policy of the VTMs is specified within this template as shown in figure 18. Furthermore, a task template specifies the associated tasks, the abstract resources with their related management policies, and the importance of each VTM. A high importance value is assigned to critical tasks whereas lower importance values are assigned to tasks where contention does not have a drastic impact in the system. In addition, sub-tasks inherit importance values from their super-tasks. There is also a default VTM that includes all those activities that are not represented in the resource model. That is, such activities would use the resources defined by the default VTM. The mapping of QoS values to resource parameter values may be achieved by mathematical translation or trial-and-error estimations as described in (Nahrstedt et al., 1998).

As an example of the definition of abstract processing resources, consider the particular instantiation of the resource framework whereby VTMs encompass both a team and a buffer abstraction, as shown in figure 19. The scheduling parameters of the VTM are defined which include execution time, period and CPU usage. The team abstraction is then specified in terms of a particular number of threads. Furthermore, the definition of threads include their scheduling policy along with their thread priority. The amount of buffer allocated is also defined together with its management policy. It should be noted, however, that the language is not restricted to these types of resources or the abstraction levels and can be extended to cover a different instantiation of the resource framework.
A resource of any type may be shared between two or more VTMs as long as they live in the same address space. Allocation of shared resources is defined within the task template as shown in figure 18. In particular, the shared resources clause specifies both a list of the shared abstract resources and a list of the VTMs sharing these resources. This approach is useful in cases where, for performance reasons, it is necessary for tasks to share resources but still have certain resources allocated only to them. An example of such a case is a protocol stack in which each layer corresponds to a different task, as shown in figure 20. In this stack, each layer has a memory buffer allocated for its own use. However, it uses the same thread along the stack to avoid thread switches in order to achieve a good performance of the system.

**Figure 18. Definitions in TDL**

**Figure 19. Example of the Specification of Abstract Resources.**
Finally, the join points for this aspect language are related to the execution of the VTM that supports the execution of a task. More concretely, these join points are the invocation of task switching points. The access to a join point causes the activation of the appropriate VTM in order to support the resource demands of the task.

![Figure 20. Example of a Protocol Stack Sharing CPU Resources](image)

### 4.5 Resource Description Language

The RDL, also called the resource template, describes the platform-dependent resource requirements for the execution of a task instance as shown in figure 21. Whereas the TDL defines the resources provided for the execution of a task, the RDL defines the platform specific requirements for the execution of one or more object methods. Thus, a resource template concerns the specification of aspects such as the worst-case-execution time, typical execution time, execution period, amount of memory buffer and network resources. There is also an RDL description per address space as the information provided is platform specific. The join point for this aspect language is the invocation of a service type. Hence, the activation of a join point brings in information about the resource requirements of a task instance for the purposes of performing the admission control test and reservation of resources. Thus, whenever a service is required, an admission test is performed on the basis of the resource requirements specified by the RDL. If successful, such resources are then reserved.

<table>
<thead>
<tr>
<th>Resource requirements =</th>
<th>“Resource Template”:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{task requirements}</td>
</tr>
<tr>
<td>task requirements =</td>
<td>“Task”; task name, “:”,</td>
</tr>
<tr>
<td></td>
<td>{specific requirement}</td>
</tr>
<tr>
<td>specific requirement =</td>
<td>[cpu], [buffer], [other resource type]</td>
</tr>
<tr>
<td>cpu =</td>
<td>“CPU”;</td>
</tr>
<tr>
<td></td>
<td>“worse case time:”, digit,</td>
</tr>
<tr>
<td></td>
<td>“typical time:”, digit,</td>
</tr>
<tr>
<td></td>
<td>“period:”, digit;</td>
</tr>
<tr>
<td>buffer =</td>
<td>“Buffer:”, digit;</td>
</tr>
</tbody>
</table>

**Figure 21. Definitions in RDL**

As an example of RDL descriptions, consider the specification of the resource requirements for the execution of an instance of the task transmitAudio_V1.unmarshall as depicted in figure 22.
This task demands 10 ms of CPU time in the worse case whereas the normal operation of the task requires only 5 ms. The task also requires to be executed every 120 ms and demands 200 KB of memory for the purposes of buffering (again, the specification of resource requirements is not restricted to these two types of resources, thus, other type of resources may be included such as network bandwidth, storage resources and battery life).

<table>
<thead>
<tr>
<th>Resources Template:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task transmitAudio_V1.unmarshall:</td>
</tr>
<tr>
<td>CPU:</td>
</tr>
<tr>
<td>worse case time: 10</td>
</tr>
<tr>
<td>typical time: 5</td>
</tr>
<tr>
<td>Period: 120</td>
</tr>
<tr>
<td>Buffer: 200</td>
</tr>
</tbody>
</table>

Figure 22. Example of RDL Descriptions

4.6 QoS Management Graph Description Language

The QMGDL is responsible for defining the QoS management structure of the system. Hence, Xelha definitions of dynamic QoS management are mapped to QMGDL descriptions. As in Xelha, monitors and strategy selectors are represented by an automaton. The strategy activator is then represented as an object. In addition, event collectors are represented as object methods in charge of observing the behaviour of the object. These operations are accessed through a particular interface of the object.

<table>
<thead>
<tr>
<th>QoS management =</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;QoS management graph Template:&quot;;</td>
</tr>
<tr>
<td>{qos graph}-;</td>
</tr>
<tr>
<td>qos graph =</td>
</tr>
<tr>
<td>&quot;Def QoS graph for&quot;, service type</td>
</tr>
<tr>
<td>&quot;interfaces:&quot;,</td>
</tr>
<tr>
<td>{interface}-,</td>
</tr>
<tr>
<td>&quot;edges:&quot;,</td>
</tr>
<tr>
<td>{edges}-;</td>
</tr>
<tr>
<td>interface =</td>
</tr>
<tr>
<td>interface name,&quot;;&quot;,&quot; component name} connector name,&quot;;&quot;,&quot; interface name</td>
</tr>
<tr>
<td>edges =</td>
</tr>
<tr>
<td>&quot;(&quot;; interface name, &quot;,&quot;, interface name, &quot;)&quot;;</td>
</tr>
</tbody>
</table>

Figure 23. Definitions in QMGDL

As shown in figure 23, QoS management graphs are defined either on a per service type basis or on a per service basis. The former represent a broader case in which all services within a service type follow the same pattern for the QoS management graph. In contrast, the latter represents a more specialised case whereby the QoS management structure of a service follows a different pattern from the other services defined in the service type. For instance, the QoS management of the service Audio_Comm_V3 whose service type is AudioCommService is separately defined, as shown in figure 24.
Finally, the join points for this aspect language are the invocation of the constructor of the class that implements a service. The activation of a join point then results in the instantiation of the associated QoS management graph. Hence, the automaton and activator are connected to the appropriate object interfaces of the service.

5. Implementation

5.1 Overview

We have implemented a tool in Python that interprets both Xelha specifications and RCDL definitions. The interpretation process of the tool has basically two phases as shown in figure 25.

![Diagram](image)

**Figure 25.** Processing Xelha Specifications.

The first phase concerns the interpretation of Xelha specifications. As a result of this interpretation process, RCDL descriptions are generated which involve various aspect programs. The second phase is then concerned with the processing of RCDL definitions and the instantiation of the resource framework.
The following sections describe the implementation details of both phases.

5.2 Phase 1: Processing Xelha Specifications

The first phase is in charge of parsing Xelha specifications and generating both object classes in Python and lower-level descriptions of the configuration of resources in RCDL. For the generation of Python classes, both components and connectors are mapped to object classes. An example of such a class is shown in figure 26.

Regarding the generation of RCDL descriptions, at present there is only support for the generation of TSDL programs and partial support for the generation of SDL definitions. TSDL descriptions specify task switching points in terms of the object class, the interface and the method. The latter is associated with a task or with an “if” statement in case of a multiple task switching point. An example of a TSDL program generated by the tool is depicted in figure 27.

```python
# File generated by the Tool
import lbind
import opbind
import composite
import component
import streambind
class AudioConnector_V1(compositeComposite):
    def __init__(self, lcap, rcap):
        interfaces = {"IN": ("srcStub", "IN"), "OUT": ("sinkStub", "OUT")}
        componentGraphSpec = {
            "comps": {
                "sinkStub": {'factory': component.componentFactory, 
                              'args': ("IN", "CTRL", "OUT"), 'lcap'},
                "srcStub": {'factory': component.componentFactory, 
                             'args': ("IN", "CTRL", "OUT"), 'rcap'},
                "streamConn": {'factory': StreamConnector_V1_0, 'args': (rcap, lcap, 
                                                                        'IN'), 
                              'CTRL': lbind.lRef(None, "%put"), 
                              'OUT': lbind.lRef(None, [], "%put"))
            },
            "ifaces": {
                "OUT": ("srcStub", "OUT"),
                "streamConnIN": ("streamConn", "IN"),
                "streamConnOUT": ("streamConn", "OUT"),
                "IN": ("sinkStub", "IN")
            },
            "edges": {
                ["OUT", "streamConnIN"],
                ["streamConnOUT", "IN"]
            }
        }
        compositeComposite.__init__(self, interfaces, componentGraphSpec)
```

**Figure 26.** Generated Object Class for the AudioConnector_V1 Component Type.

Within a SDL description, component services offering QoS are represented as service type definitions in terms of the supported QoS region, the associated task and the object class that implements the service. Information about QoS regions is not provided by the tool, however, the programmer can manually define such regions. An example of a SDL program generated by the tool is shown in figure 28.
TDL and RDL descriptions are currently hand-coded (type checking and automatic QoS specifications translation concern future work). It is important to mention that the generation of Python classes define how the objects that implement a service are interconnected. This configuration includes the QoS management structure that provides the adaptive behaviour of the system. Thus, at present no support is offered for the generation of QMGDL descriptions which define such management structure.

The TA tool suite described in (Blair et al., 1999b; Blair and Blair, 2000) can be used to model the timed automaton for the elaboration of the QoS management structures. This tool converts automaton graphs into FC2 files (Madelaine and Simone, 1994). Notably, the tool also supports the composition of TA. Therefore, the monitor and the strategy selector are defined separately and later on joined together in a single automaton. Afterwards, the automaton is translated into an FC2 file and an FC2 processor tool (Andersen et al., 2000a) translates this file into Python object classes.

5.3 Phase 2: Processing RCDL definitions

The second phase concerns the processing of RCDL descriptions, which takes place at load-time. As a result of this interpretation process the three hierarchies of the resource model are instantiated. The instantiation of such hierarchies involve two important issues. Firstly, the pools of resources (i.e. VTM) for the specified services are allocated as a result of weaving the TDL program. The weaving process is performed at load-time and also involves the creation of a registry of the mapping between
tasks and VTM. This information is essential for the join point of this aspect language to select the VTM supporting the execution of its associated task.

Secondly, both task interceptors and buffer interceptors are placed in the object configuration. This is performed as a consequence of weaving the TSDL aspect program. In addition, a registry of the mapping between object methods and tasks is created. The weaving of this language is carried out at run-time. More precisely, the weaving is performed at object creation-time. For this purpose, objects within the system inherit from the \texttt{Object} class. Every time an object is created, the constructor of the \texttt{Object} class asks the VTM factory, by invoking the method \texttt{mapTasks()} of the factory, to map tasks to the object that is being created. The mapping is accomplished on a per method basis and is placed in a registry located in the interface meta-space related to the object method. The purpose of this record is to provide a means of accessing the resources (i.e. VTMs) a method is related to.

The method \texttt{mapTasks()} is also responsible for the creation of interceptors as an attribute of the object’s interface. In case of an exported method a task interceptor is defined. For an imported method, in contrast, a buffer interceptor is created. At present no support is provided for weaving the rest of the aspect languages. Hence, the join points for both the SDL and the RDL are manually coded. However, the information provided by these languages is processed by the tool. That is, interpretation of the RDL program populates a registry of the resource requirements demanded by task instances. This registry is placed in the resources meta-model. The information provided by the registry is used for collecting resource demands of a service when performing admission control tests.

Finally, the processing of SDL definitions generates a registry of the supported services. This registry provides information about the level of QoS, the task and the object class that are associated with a service. This record is inspected by the QoS manager to select the service that best matches the required level of QoS when a service type is requested. As a result, an instance of the class that implements the service is created if its associated VTM has enough resources to support this level of QoS.

6. Evaluation

This section presents an evaluation of the Xelha and RCDL languages. A series of experiments were carried out using a Python prototype of both a the OpenORB middleware architecture (Andersen et al., 2000b) and the resource framework (Duran-Limon and Blair, 2002). In particular, an audio application middleware service (Duran-Limon, 2001) was specified in Xelha and then converted into RCDL definitions by the interpreter tool. These definitions were further tuned and then translated at load-time for the middleware platform along with the audio service to be instantiated. Further details on the specification and implementation of the audio service can be found in (Duran-Limon, 2001).

The evaluation is based on the experience obtained from the experiments mentioned above. In particular, we evaluate the description languages with respect to their suitability for the specification of both the configuration and reconfiguration of QoS management in multimedia middleware, as defined in section 2.2.3.
i) **Level of specification.** Xelha combines the features provided by both ADL and the task model. Both of them offer a high-level of modelling. As a result, a high-level specification of distributed real-time systems is feasible. In addition, the RCDL allows the user to specify low-level details of the system, which Xelha is not able to define. Examples of these details include the specific platform-dependent resource requirements of a component and the implementation details of adaptive behaviour.

ii) **Separation of concerns.** The ADL is divided into several sections whereby each one of them is related to a different concern. In particular, different sections are defined for the specification of components, connectors, interfaces, the composition graph, tasks and services. Furthermore, the RCDL consists of a series of aspect languages, which offer a separation of concerns among aspects that cross-cut each other. As a result, a more readable and reusable specification is obtained.

iii) **Expressiveness.** Xelha provides support for distributed multimedia systems. Regarding the definition of QoS properties, Xelha covers the most important QoS categories for multimedia, namely timeliness, volume and reliability properties. Moreover, Xelha also provides support for both static and dynamic QoS management. As multimedia systems are typically dynamic, Xelha also supports the specification of dynamic architectures. On the other hand, the RCDL is rich enough to represent both the resource model and the task model. Hence, the RCDL can be used to specify the services provided by a distributed multimedia system whereby different services featuring distinct levels of QoS may be defined.

iv) **Tool support.** Currently, only partial tool support is offered for Xelha. In particular QoS translation and QoS analysis are not supported by the tool. In addition, no type checking mechanism is provided. Regarding the RCDL, the tool currently provides support for the interpretation of the following languages: SDL, TSDL, TDL and RDL. However, support is only offered for the weaving of TSDL and TDL languages. Hence, the tool offers partial support for both the interpretation and the weaving of the aspect languages.

<table>
<thead>
<tr>
<th>Evaluation of the description languages</th>
<th>Xelha</th>
<th>RCDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level of specification</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Separation of concerns</td>
<td>met</td>
<td>met</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>met</td>
<td>met</td>
</tr>
<tr>
<td>Tool support</td>
<td>partly met</td>
<td>partly met</td>
</tr>
</tbody>
</table>

**Table 2.** Fulfilment of Requirements for the Description Languages

Xelha allows for the high-level specification of distributed multimedia systems whereas the RCDL language supports the definition of lower-level resource management aspects. The use of these languages facilitates the development of dynamic resource management support
for distributed multimedia applications. A summary of the evaluation of the description languages is presented in table 2.

Regarding the costs of using the languages we can mention that since neither Xelha nor RCDL support extensibility, a different language interpreter has to be implemented if any of the languages are modified. There are two main cases in which the languages need to be changed or extended: a) when other QoS categories than those defined by Xelha are used, and b) RCDL might need to be modified every time a different resource framework instantiation is deployed. However as pointed out in section 8, future work is considering overcoming this issue in order to cover other real-time domains.

Finally, we have also carried out a number of experiments to test the performance of the middleware platform. Note that as this paper is mainly focused on the language support for multimedia middleware and not the middleware platform itself, we do not present here the experimental results obtained. However, the results can be accessed elsewhere as indicated below. The use of reflection introduces some overhead (Duran-Limon, 2001) that has to be taken into account when designing a multimedia application. The performance of the prototype (Duran-Limon, 2001; Duran-Limon and Blair, 2002) could be improved by either using other mechanisms provided by Python with less overhead or developing specialised modules in C that provide an efficient version of the required functionality. Indeed, the case study presented in (Duran-Limon, 2001; Duran-Limon and Blair, 2002) has demonstrated the feasibility of both resource introspection and resource reconfiguration in middleware platforms. That is, despite the performance penalties introduced by the Python prototype, the case study demonstrates that our prototype provides sufficient support for small-scale multimedia applications. A more efficient implementation of the resource framework could be used for larger applications.

7. Related Work

The work presented here is related to various areas. Some of the most important related work is briefly discussed below.

7.1 Middleware

TAO (Schmidt et al., 1998) is a high-performance real-time ORB. TAO is CORBA-compliant and has introduced several extensions to this standard in order to overcome the shortcomings of conventional ORBs regarding efficiency issues. QoS specifications are defined by populating the structure \texttt{RT\_Info} on a per schedulable operation basis. The specifications of the operations’ QoS attributes include criticality, worse case execution time and period.

CORBA allows for the specification of QoS by employing pre-defined IDL interfaces. In particular, the Messaging specification (OMG, 2001b) offers facilities for the customisation of some QoS aspects of the ORB. Examples of such customisations are the level of reliability of a one-way message, request timeouts and the maximum number of hops a request may go through. Resource configuration of the ORB can then be achieved by the use of the Real-time specification (OMG, 1999). Such a configuration includes thread pools and memory buffers. Network protocols may also be selected and their protocol properties configured. In addition, the portable object adapter (POA) (OMG, 2001a),
allows the developer to set the threading policy at POA creation time. Furthermore, Portable Interceptors (OMG, 2001.) enable the customisation of the IOR (Interoperable Object Reference) at creation time whereby QoS information related to security, server thread priorities and network connections may be defined. Although the two approaches above provide important support for the definition of QoS management, they provide a lower-level solution which can be tedious and error-prone (in contrast to the more architectural approach offered by Xelha).

The proposal of the OMG for a UML Profile for Scheduling (Selic, 2000) covers a resource framework in which both physical and logical resources may be represented at various levels of abstraction. Examples of the former are processors, memory and networks, whereas the latter include buffers, queues and semaphores. A resource is viewed as a server with associated QoS attributes in charge of attending client demands. Moreover, a resource may provide one or more services. A QoS contract then captures both the client’s QoS requirements and the QoS offered by the server. Nevertheless, Xelha, as an ADL, provides a more natural approach and a higher-level of specification of component configurations graphs in multimedia middleware.

The 2K\textsuperscript{Q+} framework (Wichadakul et al., 2001) provides support for the QoS specification and compilation for a reconfigurable component-based middleware. The framework offers QoS management support for multiple application domains. The approach allows the developer to specify the component configuration in a functional graph. In addition, the service component description template permits the developer to specify the components’ information such as the component’s type and its resource requirements. The user-to-application-specific template is used to define the mapping between user-level QoS and lower-level QoS. Although the functional graph description is close to the specifications provided by an ADL, the semantics of an ADL are generally richer. In addition, our task model enables for a higher-level of specification whereby a single statement may be used for the QoS management definition of a whole component configuration.

### 7.2 Architecture Description Languages

There has been some interest in the development of ADLs for distributed systems such as Darwin (Magee et al., 1995) and C2 (Medvidovic et al., 1996). In addition, the Aster Project has developed an ADL for the systematic synthesis of middleware configuration (Zarras and Issarny, 1998). However, none of them covers descriptions for resource management. There is, though, some related work in the area of real-time systems. MetaH (Binns et al., 1996) is an ADL for the guidance, navigation and control domain, but it is only suitable for multi-processor system architectures. UniCon (Shaw et al., 1995) is an ADL that offers support for schedulability analysis. Our task model, however, provides a higher-level of resource management. In addition, the previous work does not deal with dynamic QoS management. ObjectTime (Lyons, 1998) is a tool that implements UML for real-time constructs (Selic and Rumbaugh, 1998). However, no means is provided to model resources.

There has been some work regarding specification of dynamic architectures. For instance, Darwin (Magee and Kramer, 1996) provides two main mechanisms for describing dynamic structures, namely, lazy instantiation and direct dynamic instantiation. In the former, a component is not instantiated until the service it provides is accessed. The latter creates a replica of a component every time the service it
provides is accessed. Our approach is similar but more flexible than that taken by Darwin. That is, in our work, on accessing the service provided by a component a sub-type of any specified component type is instantiated without constraining this to the instantiation of the accessed component. Wright (Allen, 1997) also offers support for dynamic change using events, e.g. when an event occurs it triggers some actions like unbinding and rebinding components. All these approaches require that architectural changes be expressed at design time. In contrast, the ArchStudio tool (Oreizy et al., 1998), which is based on the C2-style, supports unplanned run-time modifications. More specifically, this tool supports operations for the addition and removal of components. The tool also provides an architectural constrain mechanism that validates changes to leave the application in a consistent state. However, when analysing the modifications an architecture might experience, it is easier to do so from languages that explicitly express changes.

7.3 Aspect-Oriented Programming

The Quality Object (QuO) project (Zinky et al., 1997; Pal et al., 2000; Sydir et al., 1998; Loyall et al., 1998; Venegas et al., 1998) provides a framework for the specification of QoS for CORBA object interactions. QoS aspects are specified in QDL (Pal et al., 2000; Loyall et al., 1998), which is an extension of the CORBA IDL. Notably, various description languages are defined within QDL by following aspect-oriented programming techniques. A contract description language (CDL) specifies QoS contracts. Hence, CDL describes the QoS required by clients, the QoS provided by objects, the operating regions along with the associated adaptive behaviour when transitions occur, and system conditions that need to be monitored. A structure description language (SDL) specifies the adaptive behaviour of delegates (i.e. proxy objects). A connector setup language (CSL) defines how the QuO objects are associated with the client and the object. Our approach is similar to the QuO framework, however, there are some differences between the two approaches: a) QuO focuses in CORBA-based systems whereas our framework targets a more general open framework, b) QuO’s aspect languages may only specify QoS management for fine-grained interactions, i.e. client-object interactions, whereas, our framework’s aspect languages offers support for both fine- and coarse-grained interactions and c) QoS contracts in QuO are defined in terms of operating regions whereas our approach uses tasks and management objects.

Finally, the aspect orientation paradigm is gaining acceptance in the research community and approaches for the design of configurable middleware systems (Jacobsen, 2001) and the aspect-oriented design of operating systems (Netinant et al., 2000; Coady et al., 2001) have been suggested elsewhere.

8. Concluding Remarks

We have presented an approach that allows for both the high-level specification and the low-level customisation of QoS management in multimedia middleware. High-level specifications are achieved by the use of the ADL, Xelha. Both static and dynamic QoS management properties can be defined within the language. The former are defined in terms of task graphs and QoS specifications whereas the latter are defined by management components. Importantly, monitors and strategy selectors, which are
modelled as timed automata, can be simulated and formally verified by associated tools. RCDL descriptions are then concerned with the low-level details of the implementation system, such as the amount of memory and CPU required for the execution of a component in a particular platform. Xelha specifications are mapped down to RCDL definitions with the help of tool support. The aspect separation of the RCDL allows for the easy analysis and customisation of the specifications of the resources subsystem.

The main purpose of the task model, which is supported by both the Xelha and RCDL languages, is to provide a higher-level of analysis and design of QoS management. This model enforces a clear separation of concerns by defining QoS statements on a per task-basis. Also important is the support provided by the RCDL for the resource model whereby different types of resources may be represented at various levels of abstraction. Xelha also covers the specification of dynamic architectures. That is, it allows us to define a range of type values, included in a type hierarchy, that may be dynamically instantiated as a result of accessing a service provided by a component. Although the paper is mainly focused on middleware, our approach is equally applicable to the application level.

A tool has also been implemented for the interpretation of Xelha specifications, which generates Python object classes and resource descriptions in RCDL. Further work is required for the support of both QoS analysis and QoS translation of Xelha specifications. That is, the feasibility of the QoS properties must be assessed and these properties have to be translated into specific resources and resource management policies. The language also requires support for type checking. For instance, tool support is required to ensure that a particular binding type is connected to its corresponding interface style. In addition, although type checking helps to ensure that an architecture maintains consistency after performing changes, this is not enough. Further mechanisms have to be developed to ensure valid states when modifications are performed. It would also be of great benefit to provide support for visual development which would allow for rapid prototyping. Moreover, the integration of the tool support for modelling timed automata would be of great value providing a more complete tool to the user. Although Xelha focuses on distributed multimedia systems, the language is not necessarily restricted to this area and can be extended to cover other domains. One approach for realising such extensions is using XML Schemas (W3C, 2001) for the different application domains. Finally, regarding the RCDL, further development is still required for the weaving of the following aspect languages: SDL, RDL and QMGDL.

Acknowledgements

Hector A. Duran-Limon would like to thank his sponsor, the Autonomous National University of Mexico (UNAM). Particular thanks are due to Jean-Bernard Stefani and his group at CNET who have provided partial funding for this work.
References


Appendix A: BNF specification of Xelha

Xelha Specification = "Def component", component type name, ":",  
  "components": 2 * component, {component},  
  "connectors": {connector}-,  
  [{interfaces:}, {interface}-],  
  "composition graph": composition graph,  
  [{"tasks":}, {task}-],  
  [{services:}, {service type}-];

component =  component name, ":", component type name, ","capsule name;

connector = connector name, ":", connector type name, "(", source capsule name, ",", sink capsule name, ")"

component type= "Def component", component type name, "("parameters")",  
  [{"extends":}, component type name | primitive component type name] ":",  
  [{"components":}, {component}-],  
  [{"connectors":}, {connector}-],  
  "interfaces": {interface}-,  
  [{"composition graph":}, composition graph],  
  [{"tasks":}, {task}-];

parameters =  {component type name, component name},  
  {connector type name, connector name},  
  {"string", capsule name}-,  
  {interface type name, interface name}-;

connector type =  "Def connector", ":", connector style,">",  
  connector type name,"("parameters")",  
  [{"extends":}, connector type name | primitive connector type name] ":",  
  [{"components":}, {component}-],  
  [{"connectors":}, {connector}-],  
  "interfaces":  
    "interaction": 2 * interface, {interface},  
    [{"control":}, {interface}-],  
    [{"composition graph":}, composition graph],  
    [{"tasks":}, {task}-];

interface =  interface name, ":", interface type name,"("component name, ",", interface name, ")";

interface type =  "Def interface ",<">, interface style, ":", interface type name,  
  [{"extends":}, interface type name], ":",  
  "provides": {operation}, {dynamic component}, {dynamic connector},  
  "requires": {operation};

operation =  primitive type, operation name, ",(", ["in", primitive type, argument name],  
  "out", primitive type, argument name, ")";

dynamic component =  primitive type, operation name, ",(", ["in string", interface name]-,  
  "in QoSspec", qos specifications, ")";
  "inst", component type name;

dynamic connector =  primitive type, operation name, ",(", ["in string", source interface name,  
  "in string", sink interface name, "in QoSspec", qos specifications,  
  "inst", connector type name;

interface style =  "operational" | "stream" | "signal";

composition graph =  "interfaces:".
task = primitive task | composite task

composite task = "Def task", task name, "includes", 2 * task name, {task name}. ";"
(["importance": digit],
  "qos specifications": qos specifications,
  ["qos management structure": qos mgt. structure]);

primitive task = "Def task", task name, ";"
  ["importance": digit],
  "switching points": end point, ["if": task name],
  "capsule": capsule name,
  "qos specifications": qos specifications,
  ["qos management structure": qos mgt. structure]);

qos specifications = {delay}, {jitter}, {throughput}, {packet loss}, {other}

delay = "delay(" end point,"", end point,"") = ", digit,
jitter = "jitter(" end point,"", end point,"") = ", digit,
throughput = "throughput(" end point,"") = ", digit,
packet loss = "packet_loss(" end point,"", end point,"") = ", digit;

end point = component name| connector name,";", interface name,";", operation name;

qos mgt structure = "collector":
  interface name,";", interface type name, ",", component name, ",", interface name
  "timed automaton":
    automaton name,";", component type name,
  "strategy activator":
    activator name,";", component type name,
  "qos management graph":
    composition graph;

service type = "Def service type", service type name,";"
  {service};

service = "Def service", service name,";"
  "user qos": user qos,
  "tasks": {(task name, ",", digit)};
Appendix B: BNF specification of RCDL

SDL
Service description = "Service Template:",
                      {service type};

service type = "Def service type", service type name, ":", {service};

service = "Def service", service name, ":",
          "user_qos:", user qos,
          "qos_region:",
          "delay:“, minimum, "", maximum,
          "throughput": minimum, "", maximum,
          "jitter": minimum, "", maximum,
          "packet_loss": minimum, "", maximum,
          "task": task name,
          "object": object class name;

maximum = digit
minimum = digit

TSDL
Switching points = "ObjectTemplate", object class name, ":",
                   {interface};

interface = "Interface:” interface name, ":",
            {operation};

operation = "operation", operation name, ":", task name | {task name, "if", task name} -

TDL
Task description = "Def VTM:",
                   "policy:", scheduling policy
                   {vtm};
                   default vtm,
                   {shared resources};

vtm= "Task:”, task name,
     "Abstract resources:”, abstract resources | other hierarchy,
     "Importance:”, digit;

default vtm = "Default VTM:”,
             "Abstract resources:” abstract resources | other hierarchy,
             "Importance:”, digit;

shared resources = "Shared resources:”, name shared resources,
                  "Abstract resources:”, abstract resources | other hierarchy,
                  "Vtms:”, list vtms sharing these resources;

abstract resources = "Def VTM:”,
                    "execTime:”, digit,
                    "period:”, digit,
                    "usage:”, digit,
                    "Def Team:",
                    "num_threads:”, digit,
                    "Def Thread:”,
                    "policy:”, scheduling policy,
                    "priority:”, digit,
                    "Def Buffer:”,
                    "policy:”, memory policy,
                    "amount:”, digit;
RDL
Resource requirements = "Resource Template:\n    {task requirements}\n";

task requirements = "Task", task name, ":", 
    {specific requirement}\n";

specific requirement = [cpu], [buffer], [other resource type];

cpu = "CPU":
    "worse case time": digit,
    "typical time": digit,
    "period": digit;

buffer = "Buffer": digit;

QMGDL
QoS management = "QoS management graph Template:\n    {qos graph}\n";

qos graph = "Def QoS graph for", service type | service, 
    "interfaces":
        {interface}\n    "edges":
        {edges}\n
interface = interface name, ";": component name | connector name, ";": interface name

edges = "(", interface name, ";", interface name, ");"