Classification and Comparison of QoS Specification Languages for Distributed Multimedia Applications

Jingwen Jin          Klara Nahrstedt
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{jjin1, klara}@cs.uiuc.edu

Abstract—Quality of Service (QoS) is becoming an integral part of currently ubiquitous distributed multimedia applications. However, before any QoS-related mechanisms and policies, such as admission control, resource reservation, enforcement, and adaptation, can be invoked, applications need to express their QoS requirements. A considerable amount of research has been done in QoS-aware Application Programming Interface (API) design and QoS specification language development for multimedia systems.

In this paper, we present an extensive survey of existing QoS specification languages, and methodologically classify and compare them. The paper provides users with a global and insightful knowledge of this important area; knowing how to evaluate QoS languages, and what aspects are most relevant when designing new languages.

I. INTRODUCTION

As the overall computer and communication technology evolves, distributed multimedia applications are becoming ubiquitous, and Quality of Service (QoS) is becoming an integral part of them. Being highly resource (e.g., CPU, memory, and bandwidth) consuming, multimedia applications need resource management at different layers of the communications protocol stack to ensure end-to-end service quality, and to regulate resource contention for fair sharing of resources. However, before any mechanisms and policies of a QoS-aware resource management can be invoked, users and applications need to specify their QoS requirements and the corresponding resource allocations. Furthermore, they need to describe how quality of service should be scaled and adapted in cases of resource contention or resource scarcity during application runtime.

QoS involves a multitude of properties beyond the application-specific aspects, including performance characteristics, availability, responsiveness, dependability, security and adaptivity. In general, QoS specifications (1) should allow for descriptions of quantitative QoS parameters (e.g., jitter, delay, bandwidth) and qualitative QoS parameters (e.g., CPU scheduling policy, error recovery mechanism), as well as adaptation rules; (2) must be declarative in nature, that is, to specify only what is required, but not how the requirement should be carried out; (3) need to be accompanied by a mapping and compilation process to map the QoS specification to underlying system mechanisms and policies.

The main purpose of this paper is to classify and compare the existing QoS specification languages that span across several QoS layers and present diverse properties. Specifically, we aim to take a large set of existing QoS specification languages and provide a methodology to analyze and evaluate them. We will show that our divide and conquer methodology allows us to easily group QoS specification languages into classes; to analyze individual classes with respect to their properties; to study the inter-class relations; and to see potential impact of the existing languages on future improvement or development.

The remainder of this paper is organized as follows. Given so many QoS specification languages currently in existence, we first describe the methodology we use to classify and evaluate them in Section II. We first divide the languages into layers: user layer, application layer, and resource layer, according to where in the end-to-end system layers they belong to, and then further categorize them into classes, according to their properties, such as paradigm and granularity. We proceed in Sections III, IV, and V by presenting QoS specifications of each layer with representative languages. Evaluation of the languages is presented throughout the text. Mapping between the QoS specifications and underlying system resources becomes a natural topic of discussion when we deal with layered QoS. However, due to space limitations, this subject will only be presented very briefly in Section VI. Section VII provides some general comments and concluding remarks.

II. METHODOLOGY

Given the heterogeneities of the applications, user preferences, underlying operating systems, networks, and devices, and given the dynamics of their resource usages, it is a complex task, but of great importance, to properly specify QoS requirements and adaptation policies for multimedia processing and delivery. To satisfy this complex task, many QoS specification languages have emerged out of practical needs. With so many existing QoS specification languages, spanning across several QoS layers and language paradigms, there is a strong need to categorize and compare them in order to obtain a wide and insightful understanding of this important area.

This work was supported by the DARPA grant, F30602-97-2-0121, NSF grant CCR-9988199, and NASA grant NAG2-1406. Jingwen Jin is also supported by CAPES/Brazil. The work is an extension of a summer research report done in 2000.
The methodology that we will use to analyze the large set of QoS specification languages is *divide and conquer*, where we partition the set into classes according to specific properties and evaluate each class separately, as well as study the interclass relations. In each class we will present representative languages, and analyze their impact on future work. In this section, we present how we partition the languages (Section II-A), and which criteria we use to evaluate them (Section II-B).

A. End-to-end Partitioning of QoS Specification Languages

Traditionally, QoS has been a topic discussed mainly in the network communication area. However, for multimedia systems, QoS must be assured not only at the network layer, but at the end systems (e.g., OS and devices) as well. Hence, specification of QoS was introduced for end-systems and applications [1]. Based on this development, we will partition the QoS specification languages first into layers according to where in the end-to-end system they belong to [2], [3], and then into classes based on their properties.

1) Layer Partitioning: The layer partitioning is necessary as QoS specifications at different layers present very different features of services. We consider three layers: user-layer, application-layer, and resource-layer\(^1\), because the three-layer model covers QoS specifications required at all layers of end-to-end distributed multimedia systems.

1) *user-layer specification*: At the beginning, a genuine user may have the choice to specify the quality he/she expects to receive, hence we need *user-layer QoS specification*.

2) *application-layer specification*: Later on, the human-perceptive quality will be translated into more concrete QoS parameters related to the application the user is running, which we call *application-layer QoS specification*. This first mapping between user and application QoS specifications should assume no knowledge of the underlying operating system and network conditions; it should be only aware of the application alone.

3) *resource-layer specification*: Finally, in order for the application to be executed in a real OS platform and network, those application-specific QoS parameters need to be further interpreted into more concrete resource requirements, such as bandwidth/memory allocation and CPU scheduling policies. The resource management requires *resource-layer QoS specifications* to provide the QoS awareness.

Note that the main difference between an application-layer specification c and a resource-layer specification is that the former is only application-specific, c but hardware and platform-independent, while the latter depends heavily on the physical world, i.e., hardware and platform-dependent.

As a simple example to illustrate the three-layer specification model, we will consider a VOD application where a user wants to see a video located at a remote server. The user may specify, usually through a graphical user interface - GUI, the desired overall media quality for the VOD service among a set of choices, e.g., high, average and low quality, based on human perception. This specification is then passed to the application layer to derive more concrete media quality specifications such as video frame rate, image/audio resolution, and inter/intra-stream synchronization. These are application-specific, but hardware and platform-independent descriptions. Translation of these descriptions into physical or logical resource specifications for OS and communication services will be performed by the next mapping process. Operating system and communication services are specified both quantitatively and qualitatively, where quantitative descriptions may include throughput, delay, delay-jitter, buffer size, and synchronization skew, and qualitative descriptions may include CPU scheduling policies and transmission error recovery mechanisms. The former is used in a specific service as threshold parameters to classify in which range the service should operate, while the latter is used to coordinate the involved service as well as to indicate transitions from one operational mode of the service to another. A summary of QoS layers and their corresponding QoS issues is given in Table I.

2) *Class Partitioning*: QoS specification languages can be further classified into classes according to their properties. User-layer QoS specification will only be briefly presented in this paper because, compared to QoS specifications at other layers, it has been put less effort in the multimedia system QoS area. We categorize application-layer QoS specification languages, according to their paradigms, into script-based, parameter-based, process-oriented, logic, markup-based, aspect-oriented, and object-oriented paradigms, and categorize resource-layer QoS specification languages, based on their granularities, into fine and coarse granularity classes. These language classes as well as their properties will be detailed in the following sections.


Since QoS specifications at different layers present very different features of services, we evaluate each layer with different sets of criteria.

**User Layer**: A genuine multimedia application user is not expected to be a computer expert, thus it is desirable to provide the user with a user-friendly Graphical User Interface (GUI) that is simple yet expressive enough as to allow him/her to choose the most appropriate quality within a desired price range. By expressiveness, we expect that a GUI provides choices for both quality and pricing. Note that this paper concentrates on system-related QoS issues. Thus, although there is active research on graphical interface development/evaluation in the Human Computer Interaction (HCI) area, we do not discuss them, because they largely focus on aesthetic- or facility-based issues.

**Application Layer**: Among the three end-system layers, so far application-layer QoS specification have been investigated the most in the research community. We thus accordingly devote a large portion of this paper to the discussion of specifications at this layer. Five criteria will be adopted when evaluating languages at this layer:

- **Expressiveness**: We evaluate a QoS specification in terms of its capability to specify a wide variety of services, their required resources and corresponding adaptation rules.

\(^1\)Resource layer includes specifications of QoS for OS and networks.
QoS Issues/Parameters

<table>
<thead>
<tr>
<th>QoS Layers</th>
<th>QoS Issues/Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Level (subjective criteria)</td>
<td>- perceptive media quality (e.g., excellent, good, fair, bad)</td>
</tr>
<tr>
<td></td>
<td>- window size (e.g., big, medium, small)</td>
</tr>
<tr>
<td></td>
<td>- pricing model (e.g., flat rate, per transmitted byte charge)</td>
</tr>
<tr>
<td></td>
<td>- range of price (e.g., high, medium, low)</td>
</tr>
<tr>
<td>Application Level (hardware- and platform independent)</td>
<td>- application-specific QoS attributes (e.g., video frame size, video frame rate, image/audio resolution, inter/intra-stream synchronization)</td>
</tr>
<tr>
<td>Resource Level (hardware- and platform-dependent)</td>
<td>- quantitative issues (e.g., throughput, delay, delay jitter, memory size, timing of resource requirements)</td>
</tr>
<tr>
<td></td>
<td>- qualitative issues (e.g., OS scheduling, reservation style, loss detection/recovery mechanisms)</td>
</tr>
<tr>
<td></td>
<td>- adaptation rules</td>
</tr>
</tbody>
</table>

TABLE I

A SUMMARY OF QoS ISSUES FOR THE THREE LAYERS: USER LAYER, APPLICATION LAYER, AND RESOURCE LAYER.

- **Declarativity:** A QoS specification should be declarative in nature, thus should only specify what is required, not how a specification is carried out. By stating QoS requirements declarative, applications are relieved from the burden of coping with complex resource management mechanisms needed for ensuring QoS guarantees.

- **Independence:** We evaluate if specifications can be developed independently from functional code for readability and for ease of development/modifyation purposes.

- **Extensibility:** We analyze how easily we can extend a language to allow specifications of new QoS dimensions other than those already developed.

- **Reusability:** Reusability becomes important when QoS specifications get large. Sometimes it may be the case that a new specification is just an existing one with some minor refinements. Thus a language with reusable features would be favorable in practice. One necessary, but not sufficient, condition for achieving reusability is independence, i.e., separation of non-functional QoS specifications from traditional functional features. One obvious reason for this is that the separation would make both functional and non-functional code clean, easy to develop, and easy to maintain. Besides, separation also allows a single application to be associated with different QoS specifications at users’ request.

**Resource Layer:** At the resource layer, descriptions about the exact physical resource requirement (amount of resource as well as timing of allocation) and adaptation policies are expected. We evaluate resource-layer QoS languages using the expressiveness criterion.

III. USER QoS SPECIFICATION

A. Characteristics

Multimedia users need to have access and capability to control and customize the quality of their multimedia applications. One common way is to provide users with a graphical user interface(GUI) that is simple, because users are not expected to give sophisticated descriptions about QoS requirements. Therefore, giving a user-friendly GUI with a limited number of options that concentrate on subjective user-relevant quality criteria is desirable. There are two main features that user-layer QoS specifications should satisfy: (1) provision of perceptive media quality descriptions (e.g., excellent, good, fair or bad quality) and other related specifications, such as window size (e.g., big, small), response time (e.g., interactive, batch), and security (e.g., high, low); (2) provision of service pricing choices, i.e., users should be able to specify the range of price they are willing to pay for the desired service. Cost of service is an important factor, because if there is no notion of cost involved in QoS specifications, there is no reason for the user to select anything other than the highest level and quality of service [5].

B. Case Studies

An interesting study of user-layer QoS specifications was done in the INDEX project [4]. The INDEX QoS architecture captures, via an intelligent agent2, the user’s preferences in terms of service quality and price, and maps them into corresponding QoS of Internet network services so that the user’s cost-performance relation is optimized. INDEX provides pricing and media quality descriptions via four GUI panels, which reflect billing information, usage information, user preference (price selection and user feedback), and user complaints. The specified QoS information is then used by the underlying agent communication module to select an appropriate ISP (Internet Service Provider) to carry out the data communication.

Another user-layer QoS specifications can be found in the QoStalk project [6]. The QoStalk framework represents a QoS programming environment for distributed multimedia applications and provides a visual programming tool to describe user-layer QoS. The visual tool allows application programmers to specify, using a hierarchical approach, application service components, where each component is labeled with corresponding application-layer QoS descriptions. Furthermore, application developers provide user-application templates, which include user-application QoS descriptions and their mapping to the corresponding application-layer QoS. A user-application

---

2 An intelligent agent is embedded into the general QoS architecture [5].
template allows to express media-quality descriptions and non-
media-quality descriptions in a simple way, hence allows for
specific cations of different application domains.

The two visual tools have different emphases: the IN-
DEX project concentrates on pricing issues, while the QoStalk
project concentrates on descriptions of general QoS issues for
multimedia applications. The combination of the two would
make the user-layer GUI more expressive or more complete.

IV. APPLICATION-LAYER QoS SPECIFICATION

Application-layer QoS specification is necessary, as appli-
cation developers recognize that in order to access the lower-
layer services, and control their qualities, applications need to
describe their QoS requirements. A specification at this layer
is application-specific, but resource independent. Two types of
application-layer features can be defined: one is performance-
specific, expressed via quantitative parameters (e.g., frame rate,
frame resolution, synchronization skew, level of security), and
the other is behavior-specific, expressed via qualitative param-
eters (e.g., what to act if network bandwidth is scarce). A speci-
fication language should provide certain abstractions so that
programmers do not have to engage themselves into every low-
level detail about how what specific c resources and actions need
to be invoked. There are two ways of providing programming
abstractions: one is through APIs, the other is through language
constructs, by either extending existing languages or creating
completely new ones. This section presents several QoS speci-
fication languages and classifies them into different paradigms.
Most of the currently existing QoS languages utilize script-
based paradigm, parameter-based paradigm, process-oriented
paradigm, logic paradigm, markup-based paradigm, aspect-
oriented paradigm, and object-oriented paradigm. Following
we describe each one in detail.

A. Script-Based Paradigm

Script languages are more abstract than imperative lan-
guages, thus they can be used to specify things in high level,
without having to reveal too much details. Roscoe and Bowen
[7] present a technique that adds QoS support into Windows NT
applications without modifying the applications themselves or
underlying operating systems. Their solution is to add a piece
of software, called protocol agent, into the Winsock protocol
stack, where the protocol agent intercepts calls made by the
application to the networking facilities provided by the under-
lying operating system, and then contacts the policy daemon,
which contains policies expressed in SafeTcl [8]3, to learn how
to mark the network packets according to what is described in
the QoS script. Two sample policies, written in SafeTcl, are
shown in Figure 1. The first example assigns a higher impor-
tance, thus a higher Type of Service (TOS), to network con-
nections initiated by an instance of NetMeeting, and the second
example assigns a higher TOS to a particular machine, whose
IP address is 199.2.53.102.

3Developed by Ousterhout [9], Tcl is a general-purpose, interpreted scripting
language built as C library package, whose original purpose was to build a
reusable command language. By hiding many of the details of C, Tcl allows
rapid development of programs. SafeTcl is an extension of Tcl with added
security issues.

The approach used in [7] was mainly to allow existing applica-
tions to take advantage of QoS facilities described by the
DiffServ framework4. Hence, SafeTcl scripts were only used to
instruct the protocol agent how to mark packets (although the
language itself is far more expressive). It remains to be seen
how well the language can be extended to specify other QoS
dimensions. This approach allows specific cations to be written
syntactically separate from the applications, thus has good in-
dependence.

B. Parameter-Based Paradigm

Many of the current QoS-aware applications, when speci-
fying application-layer QoS, use parameter-based approach.
In this approach, application developers define data structures
to express and declare qualitative and quantitative parameters,
without creating a new QoS language; it relies on the under-
lying QoS management architecture to evaluate and act on the
parameters.

An example of the parameter-based paradigm is QoS-A, de-
volved by Campbell [10], where it uses a Quality of Service
Architecture (QoS-A) to deal with QoS enforcement both at end
systems and in the networks in a uniform way. A QoS parame-
ter specification between two communicating parties defines
a service contract that includes several aspects: flow specifi-
cation, QoS commitment, QoS adaptation, QoS maintenance,
reservation style, and cost. The service contract is implemented
as a C structure, and each clause again is represented as another
structure. The service contract structure and the structure of one
of its clauses, flow specification, are shown in Figure 2.

The flow specification structure specifies performance-
related quantiative metrics, such as frame size, frame rate,
burst size, peak rate, delay, jitter, and loss rate. The QoS com-
mitment clause describes the requirements in a qualitative way,
e.g., whether the services are guaranteed, statistical, or best-
effort. The QoS adaptation structure is used to specify which
remedial actions to take in the presence of QoS violation, where
adaptation actions can be triggered upon the degradation of, for
example, loss, jitter, throughput, and delay. An action can be

4DiffServ is a proposed IETF model for offering differentiated services in
the Internet, by marking IP packets with a byte value known as the DS field
specifying how the packet should be treated on a per-hop basis by routers.
like “if the maximum end-to-end delay is exceeded then the QoS-A will inform the user of the QoS event via an upcall”. The QoS **maintenance** structure provides choices for an application to specify how frequent or how well it wants to be notified of performance changes. The **reservation styles** structure allows to specify resource reservation styles, e.g., **fast**, **negotiated**, and **forward**. Lastly, the **cost** structure allows applications to specify the range of price or payment mode that the user is willing to follow.

The set of QoS constructs is rather rich, making the API good in terms of expressiveness, and we believe new QoS constructs can be added without too much difficulty. QoS-A also allows contracts to be developed independently of the applications. Since QoS parameters and actions are specified as structures, QoS-A does not provide special facilities for specifying cation reuse.

**C. Process-Oriented Paradigm**

The process-oriented paradigm assumes the process model, where processes, as units of execution, communicate and synchronize with one another through message passing, or communication ports. Process-oriented QoS specification allows to associate QoS with communicating end ports, as well as to express negotiation of QoS constraints and monitoring of QoS between two ports.

An example of process-based QoS specification language is **QuAL** (Quality-of-service Assurance Language), developed by Florissi [11]. QuAL is extended from Concert/C, and its language constructs provide means for specifying contract negotiation of QoS constraints, specification of QoS violation handlers, and customization of QoS monitoring. QuAL supports handling of two types of QoS metrics: application-specific QoS metrics and resource-specific QoS metrics. This section concentrates on its application-specific QoS metrics. The resource-specific QoS metrics of QuAL will be presented in Section V-B.2. At application layer, metrics such as frame rate and synchronization are of interest. QuAL monitors these metrics through a function call **qual_monitor**. For example, the command **qual_monitor** can be invoked (shown in Figure 3) if an application wants to monitor the inter-arrival delay of the incoming video frame at a certain port.

QuAL allows specification of filters, which are inspectors placed in ports to check the data flow to guarantee that only complying messages are injected in the communication stream. This feature may be useful if receivers are heterogeneous, because the sender may specify several filters such as **low quality**, **med quality**, and **high quality**, and let receivers tune into one of them according to their own capacity. Some other important features of QuAL include automatic QoS violation monitoring by having the application inform the QuAL runtime which conditions identify a QoS violation, so that when a violation is detected, the runtime notifies the application of the occurrence.

QuAL has a good expressiveness, and one should be able to easily develop new QoS constructs when needed. However, specifications written in QuAL are spread across the functional code, making both parts hard to develop and maintain. In addition, the language is more instructive than declarative, which should be a natural feature of any specific cation language. Like all previous languages, QuAL does not present features for specifying cation reuse.

**D. Control-Based Logic Approach**

Control-based approaches are adopted in adaptive systems for QoS specifications of adaptive policies and flow control. Some systems use adaptive control techniques such as PID (Proportional-Integral-Derivative) controller to specify and control finer granularity when scheduling flows or tasks. Other systems use fuzzy-control techniques to specify and assist in adaptation of QoS. We present an example of the fuzzy control approach, developed by Li and Nahrstedt [12], which allows to specify adaptive QoS, i.e., actions that need to be performed if QoS changes.

The fuzzy-control QoS specification is fully supported by an underlying middleware control architecture. This architecture enforces the adaptive application to behave according to the fuzzy-control specification, and it comprises two components: the adaptor and the configurator. The adaptor makes control decisions with global awareness of application QoS requirements and resource availability of the entire system. The configurator uses the fuzzy control QoS specification and translates the normalized control decisions, generated by the adaptor, into actual parameter-tuning actions to be used during the execution of the application.
The fuzzy control approach allows applications to express QoS-aware adaptation policies and preferences in terms of rules and member functions. Rules are specified in an if-then format: “if $X_1$ is $A_1$ and … and $X_n$ is $A_n$ then $Y$ is $B$”, where $X_1$, …, $X_n$ and $Y$ are parameters corresponding to certain QoS-relevant system conditions such as bandwidth and CPU, and $A_1$, …, $A_n$ and $B$ represent actual values of parameters in the fuzzy form high, low, moderate or below average for the linguistic variable cpu demand. An example of QoS-aware adaptation rule is “if cpu is very high and rate is very low then rate demand is compress”, which tells the system to compress the data because bandwidth is very low but there is large amount of cpu available.

One limitation of the fuzzy-control approach is that it is intended to specify only actions, but not other QoS properties. Thus this approach is poor in terms of expressiveness and is not sufficient for multimedia applications in general.

E. Markup-Based Paradigm

Extensible Markup Language - XML[13], developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996, is a markup language for documents containing structured information. Structured information contains both content (words, pictures, etc.) and some indication of what role that content plays. A markup language is a mechanism to identify structures in a document, and the XML specification defines a standard way to add markup to documents. Different from HTML, where both the tag semantics and the tag set are fixed, XML defines neither semantics nor a tag set. In fact XML is really a meta-language for describing markup languages. In other words, XML provides a facility to define tags and the structural relationships between them. Since there’s no predefined tag set, there can’t be any preconceived semantics. All of the semantics of an XML document will either be defined by the applications that process them or by stylesheets[13].

Based on the XML standard, a QoS language, called HQML, has been developed in [14]. Basically, the work defined a set of QoS of multimedia applications, to allow application developers to define QoS parameters and policies. A sample HQML specification is depicted in Figure 4.

A specification starts with the tag <AppConf g>, and ends with the tag </AppConf g>. Each specification has an id associated to itself. Inside the specification, tags such as <ServerCluster>, <GatewayCluster>, <ClientCluster>, and <LinkList> are used to define the QoS requirements of the server, gateway, client machines, and the properties of the links (e.g., fixed link or mobile link). HQML allows to specify adaptation rules between the pair of tags <Reconf gRuleList> and </Reconf gRuleList>. For example, in Figure 4, the adaptation rule indicates that “when Bandwidth is very low, then the application execution should switch to specification whose id is 2”.

Interested readers should consult [14] for the detailed semantics defined by different tags shown in the Figure. A good point of HQML is that it can, due to XML’s meta-language property, be easily extended to include new QoS parameters. The language is also good in expressiveness, declarativity, and independence.

![Fig. 4. An HQML specification.](image)

However, it does not have any special constructs that facilitates the extension and reuse of existing specification.

F. Aspect-Oriented Approach

Many distributed systems are built on top of CORBA - a middleware that provides a flexible communication and activation substrate for distributed heterogeneous object-oriented computing. CORBA hides system- and network-specific characteristics of objects behind the standardized Interface Description Language (IDL) specifications, so that the objects exhibit only their functional interfaces. The fact that IDL abstracts away low-level details simplifies development and maintenance of distributed objects, but at the same time, it also makes the inclusion of non-functional features (such as QoS) into the system difficult because much of the information required to support the QoS is also hidden.

The Object Management Group (OMG) has put some effort on extending CORBA to support QoS-enabled applications[15]. However, so far there doesn’t seem to be any concrete specification language developed at OMG. CORBA IDL has been extended by Becker and Gheis [16] with constructs for QoS characterizations. However, this approach statically binds QoS characterizations to interface definition, therefore, it does not allow different properties to be associated with different implementations of the same functional interface. In the research community, two different language approaches for distributed object-base applications have been developed: (a) aspect-oriented approach, and (b) object-oriented approach.

---

5This is an approach similar to that adopted by TINA ODL[17], which also syntactically includes QoS requirements within interface definition. This way, each functional interface can be only associated with fixed QoS properties.
This section concentrates on the Aspect-Oriented approach. Section IV-G will present Object-Oriented approach in detail.

The Aspect-Oriented approach follows the Aspect-Oriented Programming (AOP) paradigm developed by Kiczales et al [18], because QoS-related tasks are examples of the so-called aspects by this paradigm. Aspects are not units of the system’s functional decomposition; rather, they are properties that affect the performance or semantics of the components in systematic ways. Using traditional programming languages, implementations of such aspects would result in tangled code with aspect-related code spreading over the program and cross-cutting the basic functional components of the system. The Aspect-Oriented Programming technology has been developed to support clean abstraction and composition of both aspects and functional components. Using AOP, an application can be decomposed into functional components and aspects, where different aspects can be programmed in different languages suitable to the tasks, and at the end a special language processor, called aspect weaver, would coordinate the co-composition by weaving all the code together to produce a single executable application.

The Aspect-Oriented approach can be found in the Quality Object (QuO) framework [19], [20], [21] developed at BBN. QuO supports QoS at the CORBA object layer by opening up distributed object implementations to give access to the system properties of the CORBA ORB and objects, and extends the CORBA functional IDL with a QoS Description Language (QDL), which allows specifications of possible QoS states, system resources and mechanisms for measuring and providing QoS, and behavior for adapting to changes in QoS.

An application developed in QuO can thus program a QoS contract (using QDL) separately from the functional code, with the contract specifying both the level of service desired by the client and the level of service the object expects to provide, as well as actions to take when the level of QoS changes. When a client calls a remote method, instead of being delivered to the ORB, the call is passed to an additional component of the QuO framework, a local delegate residing at the client side. The purpose of this step is to trigger contract evaluation, and to decide how to process the method call based upon the current values of all system conditions measuring aspects of the system’s state.

QDL consists of a set of three languages, the **Contract Description Language (CDL)**, the **Structure Description Language (SDL)**, and the **Resource Description Language (RDL)**.

CDL is used for specifying a QoS contract, which consists of four major components: a set of **nested regions**, each representing a possible state of QoS; **transitions** for each level of regions specifying behavior to trigger when the active region changes; references to **system condition objects** that gather runtime information for measuring and controlling QoS; and callbacks for notifying the client or object. The last two components are specified as contract parameters, and are usually shared among several contract instances. The nested regions, describing the relevant possible states of QoS in the system, are defined by predicates on the values of system condition objects. The regions are evaluated by the contract to determine whether they are active. If the currently active regions have suffered changes since the last contract evaluation, transition behaviors are triggered.

A QoS contract written in CDL is given in Figure 5. The example shows a client contract that can operate in two possible operating modes: **Low_Cost** and **Available**. **ClientExpectedReplicas** and **MeasuredNumberReplicas** are system condition objects indicating the client’s expected number of replicas and the actual number of replicas available, respectively. The reality region transitions are used to notify the client about changes in the number of replicas. For example, in the **Low_Cost** mode, if **MeasuredNumberReplicas** drops from **High** to **Low** then a callback **availability_degraded()** will be triggered. The negotiated region transitions are used to specify actions to take when the client changes its desired replication.

While CDL is used to describe the QoS contract between a client and an object, SDL, another language among the suite of three languages of QDL, allows programmers to specify the structural aspects of the QoS application. This includes adaptation alternatives and strategies based on the QoS measured in the system. The current version of SDL enables us to express behaviors to invoke for method calls/returns based on the current regions of contracts when the calls/returns occur. Such behaviors can have descriptions like: “when the client does not desire availability and everything is OK, i.e., the contract is in the **Low_Cost** negotiated region and the **Normal** reality region, the method call is passed to the object”, or “when the client desires higher availability than what is measured, i.e., the contract is in the **Available** negotiated region and the **Low** reality region, then an exception will be thrown”. The QuO code generator would take the normal CORBA IDL code, as well as specifications written in SDL and CDL, as the input, and weave them into a single application.

![Fig. 5. A sample CDL contract.](image-url)
As we can see, QDL is good in most of our evaluation criteria; it’s expressive, declarative, independent, and extensible. However, like the previous languages, there is no special facility for specific cation reusability.

G. Object-Oriented Approach

Until this point, the languages described in this paper did not take reusability into their designs. We borrow the terminology “Object-Oriented” from the traditional languages area to concentrate on the specific cation refinement issue that, much like the class inheritance concept in Object-Oriented languages, help to enhance specific cation reusability. Many language developers may have thought that specific cations, unlike the traditional functional code, are usually small and should not be very related to each other. However, as the QoS-awareness in multimedia applications increases, we may expect specific cations in larger sizes, and two specific cations may be closely related to each other based on their properties (e.g., one is an extension of the other with slight modifi cations).

A very good example that provides specific cation refinement features is QML (QoS Modeling Language) [22], developed at the HP Laboratories. QML offers three main abstraction mechanisms for QoS specifi cation: contract type, contract, and profile; where contract type defi nes the dimensions that can be used to characterize a particular QoS aspect, contracts are instances of contract types, and pro fi les associate contracts with interfaces and operations. These concepts are illustrated in Figure 6. Reliability and Performance are two contract types each with its own dimensions, e.g., number of failures per year, time to repair a failed service, availability of the system. The contract systemReliability is an instance of contract type Reliability with constraints associating with the dimensions defi ned in the contract type. Lastly, the profile ServerProfile is defi ned for an IDL service interface (named ServiceInterface, with two operations: operation1 and operation2). The profile le requires the two previously defi ned contracts either for the service in general (systemReliability - which should hold for all operations) or for certain specifi c operations.

The QoS refi nement features in QML are actually consequences of the facts that class inheritance allows an interface to be defi ned as a refi nement of another interface and that QoS specifi cations are associated with interfaces (via profi les). Two kinds of refi ned, contract refi ned, and pro fi le refi nement, are supported in QML. A contract B refi ned from a contract A is specifi ed as B = A refined by {...} where A is the base contract, and the contract enclosed in the curly brackets is a delta that describes the differences between the contracts A and B by specifying either those QoS properties omitted in A or replacing specifi cations in A with stronger ones. Profi les can be refi ned in a similar way, with the delta specifying new contract association to be added or strengthened in the new profile. Examples of contract refi nement and profi le refi nement are given in Figure 7, where P2 is a refi nement of P1; the interface I2 is derived from the interface I1; D1 and D2 are the default contracts of P1 and P2 respectively (which means that they apply to all entities within the interface in question); C1 and C2 are specifi cally required by the operation E. For this profile le refi nement to be effective, D2 must be a refi nement of D1, and C2 must be a refi nement of C1.

As mentioned, among all application-layer languages presented so far, QML supports specific cation reusability the best through contract and profi le refi nement. It is also good in terms of independence and extensibility. However, one limitation of QML is that for each contract type (e.g., reliability) that a profi le involves, at most one contract can be used as a default contract within the profi le. For example, the profi le ServerProfile declares that it requires systemReliability, which is an instance of the contract type Reliability, it cannot require a second in-
stance of the same contract type. It would be good if future work could allow a profile to require several contracts of the same type to be the default type and take the final result as, for example, the set of strongest constraints defined in all contracts. QML is a general-purpose QoS specification language capable of dealing with any QoS aspects (e.g., reliability, availability, performance, security, and timing) and any application domain. However, one limitation is that it largely specifies QoS properties at design time, but does not address the problem of what actions to take at runtime if the QoS requirements cannot be satisfied in the current execution environment. In this respect, QDL is more expressive than QML.

H. Comparisons of Application-Layer QoS Specification Languages

We've presented different application-layer QoS specification languages so far, each with a representative example language currently existent in the literature. To better view a high-level picture of these languages’ performances according to our evaluation criteria, we provide a simple table in this section (Table II), where each evaluation criteria is attributed three values: good, fair, and poor.

V. RESOURCE-LAYER QoS SPECIFICATION

Application-layer specifications only state requirements in a rather high-level, abstract way. Later, these requirements should be further translated into more concrete resource demands. That is, descriptions such as which physical resources will be needed for the application, when they need to be allocated, which mechanisms should be adopted, and which transport protocol is to be used, need to be provided.

We classify specifications at this layer according to their granularity. Two categories of granularity will be adopted: coarse granularity and fine granularity. By coarse granularity, we only expect a meta-level specification, while by fine granularity, we expect concrete descriptions of required resources.

A. Coarse-Granularity Resource-Level QoS Specification

1) Characteristics: Some resource-layer QoS specifications only specify resource requirements in a rather abstract way. For example, they may specify what (amount of) resource is required, but do not care about when the resources need to be allocated, or what action to take if the resource requirement cannot be met, or if several resource instances (e.g., processors) are available, which specific one to use. We call languages that do not allow descriptions of fine granularity resource requirements coarse-granularity languages.

2) Case Studies: We examine two examples of coarse-granularity resource-layer QoS specification languages: RSL and SPDF.

   RSL: The Resource Specification Language (RSL) is developed by the Globus project [23][24] and is used to communicate requests for resources between components in metacomputing systems. The authors developed a hierarchical resource management architecture comprising several components, namely, resource broker, resource co-allocator, local resource manager, and an extensible resource specification language - RSL.

   Initially, an application specifies its QoS requirement in RSL. This specification is of high-level in that the required items may be physically distributed in several locations and systems. This high-level specification is passed through resource brokers that can translate it into more concrete resource requirements and locate required resources. This translation generates a specification, which the authors called a ground request, in which the locations of the required resources are completely specified. A co-allocator is then responsible for coordinating the allocation and management of resources at multiple sites, by breaking the multirequest (involving resources at multiple sites) into several requests and passing them to the appropriate local resource managers. The syntax of RSL is very simple. An RSL specification is constructed by combining simple parameter specifications and conditions with logic operators &., |, and +. As an example, the multirequest shown below specifies that the executable program, myprog, requests 5 nodes with at least 64 MB memory, or 10 nodes with at least 32 MB memory.

   \( \&(\text{executable=myprog}) \left( (\&(\text{count}=5)(\text{memory}>64)) (\&\text{count}=10)(\text{memory}>32) \right) \)

   A ground request that results from the interpretation of brokers would further specify information about which resource manager will be handling particular requirements in the multirequest, so that a co-allocator can determine to which resource manager each component of the multirequest should be submitted.

   SPDF: SPDF (Simple Prerequisite Description Format) is another very simple resource description language. It was developed as a part of the PhD thesis of Kon [25] for specifying prerequisites of application components. An SPDF specification is divided in two parts: hardware requirements and software requirements. The author considered three kinds of information in a prerequisite specification: the nature and capacity of the hardware resources, and the software services that a component requires. An example (which is self-explanatory) of SPDF specification is shown in Figure 8.

   As we can see, neither RSL nor SPDF deals with timing of resource allocation or resource scaling/adaptation. Since multimedia applications are very sensitive to timing, and are adaptive in a sense that resource requirements are usually flexible, instead of rigid, a coarse-granularity language does not suit well for multimedia applications.
### B. Fine-Granularity Resource-Layer QoS specification

1) **Characteristics:** For multimedia services, specifications of finer granularities are required. We expect from a fine-granularity resource-layer QoS specification descriptions of (1) quantitative and qualitative QoS requirements; (2) timing of the resource requirements, i.e., for when and for how long the resource needs to be allocated; and (3) adaptation rules.

2) **Case Studies:** Three examples will be presented in this section: DSRT, QuAL, and IntServ/DiffServ.

#### DSRT:
DSRT (Dynamic Soft Real Time System) is a system developed by Chu [26]. Specifically, Chu develops a mechanism to support soft real-time applications in traditional time sharing systems by developing a middleware between applications and the operating system. This new layer consists of some APIs developed in C++ that allow applications to reserve and free CPU resources. The APIs also define some structures that allow users to specify the amount of CPU resources required during the application execution (e.g., period, peak processing time, burst tolerance), and some adaptation strategies (e.g., upper and lower bounds on the guaranteed parameter that can be adjusted by the DSRT system). Once these values are set, a function call `cpu.reserve(reservation)` and `cpu.setAdaptStrategy(strategy)` will, respectively, make CPU reservation and set some adaptation strategies according to the values specified in the `reservation` and `strategy` structures. The author also extended the same mechanism from CPU to other resources such as memory and communication.

#### QuAL:
QuAL [11] provides a range of QoS attributes for the specification of network and OS-resource-layer QoS metrics. As an example, imagine a situation where an application periodically sends images to a remote site **B**. The applications at both sides may specify QoS constraints on the transmission of images as in Figure 9. Having both sites specified their requirements, QuAL abstracts QoS negotiation between peer applications by type checking connecting ports and guarantees that two ports are connected only if they have compatible QoS attributes; i.e., if the compiler and runtime are able to coerce all the QoS requirements of the sender into the QoS requirements of the receiver, or vice versa. In most cases, coercion is possible when the QuAL compiler or runtime can upgrade a less restrictive constant until it matches a more restrictive one. Coercion can be made disabled by the keyword `nocoercion`. In such a case, the input port and output port QoS constraints are compatible only if they match exactly.

QuAL also allows specification of actions to perform when QoS violations occur. The runtime automatically monitors QoS delivery and invokes application-customized exception handlers when violations are detected. The runtime scrutinizes interactions among applications, communication protocol stacks, and OS, and collects statistics on the QoS delivered into a QoS Management Information Base (QoS MIB), which is then used by applications to dynamically adjust their execution according to the QoS being delivered.

#### IntServ/DiffServ:
The vigorous interest in QoS issues within the Internet community has led the rapid development of two IP standards by IETF: the Integrated Service (IntServ) based on the Resource ReSerVation Protocol (RSVP) and the Differentiated Services (DiffServ). With the grown interest in Internet audio and video, IntServ was a standard developed around 1997 focusing on per-flow QoS, where admission control is invoked at each node to make a local accept/reject decision by the signaling protocol RSVP. The poor scalability of IntServ (due to the complex signaling protocol and state maintenance overhead incurred at routers) has, later, led to the development of DiffServ which, rather than providing QoS on a per-flow base, considers flow aggregates at the edge of the network to keep the core of the network simple. By marking packets’ type fi eld (Type of Service fi eld in IPv4 and Traffic Class in IPv6) at the edges of network, the core of the network only needs to check this type fi eld, which represents a small and well-defined forwarding behaviors, to make forward-
ing decisions. The DiffServ architecture comprises three major components: (1) a policy and resource manager for creating network policies and distributing them to the DiffServ routers; (2) edge routers for marking packets; and (3) core routers for forwarding or discarding packets. As can be seen, QoS for IP networks is largely to quantify and enforce the treatment a particular packet can expect as it transits a network. More details about IntServ and DiffServ specific cations can be found at the IETF site http://www.ietf.org/.

VI. Discussions

QoS mapping refers to the process of translating higher-level representations of QoS into lower-level representations of QoS. The translation between user QoS and application QoS is non-trivial and it is still an open research issue, because the perceptual issues are not completely understood [1], and because human perceptions for quality may change as the technology evolves.

Automated QoS mapping between application- and resource-layer can shield applications from the complexity of underlying QoS specific cations and QoS management functions. QoS mapping development between these two layers is still in its infancy. Most of the research to date has focused primarily on deriving appropriate QoS parameters for memory, CPU processing and network connections in a rather static, architecture-specific manner [10]. Individual work such as [27][28] largely deals only with partial mapping rules. They either provide only with some quantitative translation of certain parameter value into another (e.g., mapping from picture resolution to bandwidth or memory requirement), or concentrate only on mapping for specific c application based on a specific c architecture. Although QuAL[11] was intended for specifying both application and resource-layer QoS requirements, it did not deal with issues of mapping between the two layers. The newest and most significant development in QoS compilation can be found in [29].

The difficulty or discouragement related to QoS mapping development may be caused by the fact that the underlying operating systems and networks are not fully prepared to support QoS appropriately yet. For example, it makes no sense to specify advanced resource reservation at a higher level if the lower levels do not support such a feature physically. It is probable that general rules for mapping may be impossible to derive, given that there are varieties of operating systems, networks, and applications. Therefore, mapping rules are very likely to be system- and application dependent.

VII. Conclusions

In this paper, we reviewed QoS languages currently existent in the literature, and classified and compared them according to our criteria. As we can see, there are already quite a lot of specific cation languages for depicting QoS requirements in different situations. There are certainly many other languages left out from this paper due to space limitation. For example, in [30], Staehli et al defined QoS for multimedia database systems by making strong distinctions between contract, view, and quality speciation. They used the mathematical notation - Z, to denote the specific cations. Since Z specific cations are purely declarative and inherently non-executable, and since most of the Z constructs are too abstract to be refi ned to real implementations automatically by existing translation tools, such specific cations are only suitable for helping derive implementations.

There is, as yet, no consensus on the precise set of dimensions that quality of service should encompass. Much of the current effort centers on providing assurances for attributes such as cost, timeliness (e.g., response time, jitter), volume (throughput), precision, accuracy, synchronization, availability, reliability, and security.

Aspects of QoS management can be inserted into applications in a variety of ways. For example, they can be specified at the application layer while the mechanisms and enforcement are provided by the OS and communication systems; they can also be embedded into the resource infrastructure (e.g., communication network), effectively hiding from the application. Since QoS at each layer has different purposes, it will most likely become prevalent in all layers either for the purpose of ease of specific cation at high layers or for the purpose of resource enforcement at low layers. Actually, the boundary of QoS layers get somewhat blurred in some specific cation languages presented in this paper. For example, QoS-A and QuAL deal with both hardware-independent and hardware-dependent QoS parameters and adaptations. Low-level, hardware-dependent QoS specific cations are usually too complex for application developers to derive. Ideally, this should be tasks of the mapping process. Therefore, future work should give more importance to the understanding and derivation of a comprehensive QoS mapping system in order to alleviate high-level programmers from the burden of learning characteristics of low-level physical resources.

References


