Compositional Quality of Service Semantics

Richard Staehli
Simula Research Laboratory
P.O. Box 134
N-1325 Lysaker, Norway
richard@simula.no

Frank Eliassen
Simula Research Laboratory
P.O. Box 134
N-1325 Lysaker, Norway
frank@simula.no

ABSTRACT

Mapping QoS descriptions between implementation levels has been a well known problem for many years. In component-based software systems, the problem becomes how to predict the quality of a service from the quality of its component services. The weak semantics of many QoS specification techniques has complicated the solution. This paper describes a model for defining the rigorous QoS semantics needed for component architectures. We give a detailed analysis of compositional QoS relations in a video `Object-Tracker` and discuss how the model simplifies the analysis.

1. INTRODUCTION

The need for QoS management support in component architectures is well known [4]. While many aspects of QoS management have been investigated in the context of distributed systems, component architectures present new challenges. One of those new challenges is how to reliably predict QoS properties for a composition of components.

Component architectures such as the CORBA Component Model (CCM) guarantee that applications *assembled* from independently developed components will function correctly when deployed on any sufficiently provisioned implementation of the component architecture platform. We refer to this as the *safe deployment property*. Although current component standards have had good success in some domains, such as e-business, an application that performs well in one deployment may be unusable as load scales up or when connections are re-distributed across low-bandwidth connections. We use the term *QoS-sensitive application* to refer to an application that will commonly perform unacceptably if platform resources are scarce or if the deployment is not carefully configured and tuned for the anticipated load.

State-of-the-art middleware and component technologies provide QoS management APIs that force application developers to code deployment-specific knowledge into the application [1][6][8]. Rather than specifying an assembly of black-box components that will run on any standard platform, developers instead write deployment-specific configuration code that depends on knowledge of component and platform service implementations. This approach complicates development and fails to preserve the safe deployment property.

1.1 The QuA Project

The QuA project is investigating how a component architecture can preserve the safe-deployment property for QoS-sensitive applications [9]. We believe that *platform-managed* QoS is the only general solution that preserves the safe deployment property. This means that applications and application components should be written without knowledge of the runtime platform implementation and resource allocation decisions. Application specifications for accuracy and timeliness of outputs must refer only to the logical properties of the component interfaces.

Platform-managed QoS means that the platform must be able to reason about how end-to-end QoS depends on the quality of component services. The composition relationship is often non-trivial as in the case of a remote video conference where image quality may depend on both packet transport loss and video encoding.

We refer to the set of characteristics used to specify QoS as a quality model and the concepts used to define such characteristics as a QoS meta-model. To enable a component platform to reason about composition and conformance, we need a QoS meta-model that allows us to uniformly model every level of implementation as a service, from the application level down to physical resources, and that, for any service, allows us to define a quality model that does not depend on implementation. These properties allow the creation of QoS specification standards and developer provided mapping functions that can be used to derive composition QoS properties from independently developed components.

In the next section, we describe how the QuA QoS meta-model satisfies the above requirements. In Section 3 the model is applied recursively to define QoS models for a complex real-time video processing application and its components. We show how the model permits a precise mapping of component-composition QoS dependencies. Section 4 discusses related work and Section 5 presents our conclusions.

2. THE QUA QOS META-MODEL

The QuA QoS Meta-Model (QQMM) defines the semantics of our QoS specifications. It defines a generic model of a service and defines quality in a way that allows us to cre-

SAVCBS'04, October 31, 2004, Newport Beach, California, USA

ate precise and practical QoS models for any specific service type.

To begin, we need to be precise about what a *service* is. In common use, the term service means work done for another. In computing systems, we want the term to apply both to the work performed by a complex distributed application for its clients and to the work performed by the simplest of hardware resources, such as a memory address that stores a binary value for some computation.

Definition 1 A *service* is a subset of output messages and causally related inputs to some composition of objects.

This is consistent with the common definition, since the output messages represent the "work done" and the inputs represent the work request. An object's type semantics define which inputs cause which outputs.

Since network packet delivery is a service that is well known in the QoS literature, we will use it to illustrate our definitions. We can model an IP network port as a distributed port object that can accept `send(packet)` messages at multiple locations and emit `deliver(packet)` messages at an endpoint identified by the host and port number. This service is illustrated in Figure 1. We can define a service provided by this port object as consisting of only the `deliver(packet)` output events and causally related `send(packet)` input events for which the input came from source A. These packets are colored black in Figure 1.

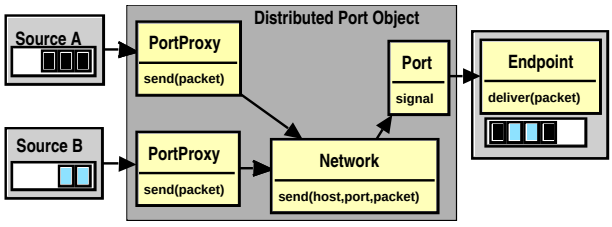

Figure 1: Example packet transport service.

In the QQMM definition of a service, we assume the most basic model of object-oriented analysis, where objects in some platform-defined identity space communicate by sending messages. The sending of a message is both an input event for the receiver and an output event for the sender. In the packet service, we can define the occurrence of the send event as precisely the moment when the send function is invoked and the occurrence of the deliver event as the precise moment when the received packet is made available in the output buffer. The QQMM makes no assumptions about object implementations and so we must allow that objects may receive input messages concurrently from multiple senders and send output messages concurrently to multiple receivers. This suits a general model of distributed computation.

As in the packet service example, other services may share the same object and even the same interface to that object, so long as they have a disjoint set of output events.

A service specification can be as simple as identifying an output interface for an object; the causally related inputs are implied by the semantics for the object type. For example, the semantics of the distributed port object dictate that every packet delivered has exactly one send event that caused it. The semantics also dictates that the value of the packet delivered should be the same as the packet that was sent.

Now, to reason about the behavior of a service, we need to talk about the history of input and output events.

Definition 2 A *message event trace* is a set of message values associated with the sending interface and the time it was sent.

A message value represents all content of the message, including its type or signature. The sending interface is the location of the mechanism used to send the message. In the packet service example, the sending interface used by `Source A` is the first `PortProxy` object, while the sending interface for packet service outputs is the `Endpoint` object. The time associated with an event comes from a local clock at the sending interface. We assume that clocks are synchronized, but acknowledge that when times from remote clocks are compared there will always be some uncertainty about how well they were synchronized.

The QQMM defines a message send to be a local and instantaneous event. All preparation for a message send is done in the sender and all processing of a message after the send event is done in the receiver. This property assures us that end-to-end processing time is consistently and properly accounted for. In the packet service example, all real delay occurs within one of the service components, including the `Network` object that encapsulates physical network access.

We use the term *input trace* to refer to the message event trace with all input events for a service, and *output trace* to refer to the message event trace with all output events for a service. The behavior of a service implementation is determined not only by the semantics of its declared interfaces, but also by the availability and scheduling of resources in the underlying platform. To define quality of service, we need to ask what is the ideal behavior of a service.

Definition 3 For a given input trace, the *ideal output trace* is generated when the service executes completely and correctly on an infinitely fast platform with unlimited resources.

That is, computation takes no time and results are obtained at the same time they are requested. Of course, events in an ideal output trace still only occur as frequently as the inputs that cause them!

Although we believe the QQMM can model QoS for nondeterministic computations, we ignore them for now to simplify the presentation. For deterministic computations, the semantics of a service's interfaces defines the ideal output trace as a function of an input trace. This means that the best possible quality for a service is always well defined.

In a real implementation of a service, the actual output trace will differ from the ideal in both the timing and value of message events. The causes for this deviation from ideal include finite CPU speed, queueing delays, and bandwidth reduction strategies. This is the stuff of QoS management.

We would like to view the possible output traces for a service as points in a behavioral space and consider distance from the ideal output trace as an error measurement, but to use this concept in QoS specifications, we must first define the dimensions of this behavioral space.

Consider again, the packet service example. Figure 2 shows the input trace as incoming messages on the left side

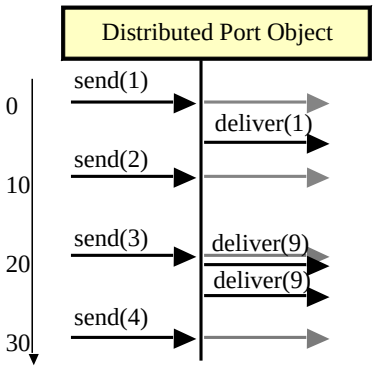

Figure 2: Ideal (shown in grey) versus actual (black) behavior.

of the service's timeline and the actual output trace as outgoing messages on the right. The ideal output trace is shown in grey. Assuming that this trace shows the entire lifetime of the service, we can see that each event in the ideal output trace can occur at a later time in the actual trace and may have its packet value corrupted or degraded in some way. The second and third packets have apparently been corrupted and it is not possible to be certain which is which without some knowledge of the service implementation. The fourth packet sent had not been delivered at the time of reckoning and may be considered lost.

The important point to observe about the packet service output is that every event can vary independently from the others in its delay from the ideal and in the change to its packet value. The QQMM generalizes from this example to make the assumption that the only possible values in an actual output trace that can differ from the ideal are the time of the output event and the values of message arguments. Each of these variables in the output trace represents an independent dimension of the behavioral space.

Let $\vec{X}$ be the vector of values that may differ in a trace X, ordered by event order in the ideal trace and by the order they occur in the message value. Then the difference between an actual output trace A and the ideal trace I is the difference vector $\vec{\delta} = \vec{A} - \vec{I}$.

For the packet service example, the values that may differ in the actual output trace are the event times and the packet values. If I is the ideal output trace for the example, the order of values in $\vec{I}$ is packet value followed by time for the first output, then for the second and so forth as shown in Equation 1.

$$\vec{I} = (1, 0, 2, 10, 3, 20, 4, 30) \tag{1}$$

If A is the actual output trace for the example, then the first two values of $\vec{A}$ are 1 and 5.

$$\vec{A} = (1, 5, 9, 21, 9, 24, 0, \infty) \tag{2}$$

As noted above, it is not possible to be certain which of the next two packets delivered corresponds to the second event in the ideal trace. This is an inescapable problem in QoS management: unreliable systems cannot be relied on to communicate enough information to unambiguously determine error. Instead, the difference vector can have many possible values depending on different interpretations of the correspondence between actual and ideal events [11]. Fortunately, it is usually safe to consider only the interpretation corresponding to the least error as it is unlikely that random faults would yield better service. So, we choose the following interpretation of $\vec{A}$ and the difference vector $\vec{\delta}$:

$$\vec{\delta} = (0, 5, 7, 11, 6, 4, -4, \infty) \tag{3}$$

Although this definition of the difference between actual and ideal allows us to define quality in a weak sense, i.e., actual traces in which every vector component is below a corresponding threshold value, it fails to tell us which of two output traces is better when each contain some components that are worse than the other. Another problem is that as the complexity of an ideal trace increases, through additional messages and more complex message structure, the number of ways in which actual traces may differ explodes. Fortunately, we frequently are concerned only with an overall measure of distance from the ideal. For example, we can ignore the individual values for event delay and instead monitor aggregate statistics such as maximum and median.

The QQMM concept of an error model provides a rigorous way to define the type of quality characteristics that are most useful for QoS management.

Definition 4 An *error model* $\vec{\epsilon} = \left(\epsilon_1(\vec{\delta}), ..., \epsilon_n(\vec{\delta})\right)$ is a vector of n functions that each map from a difference vector $\vec{\delta}$ to a real number such that $||\vec{\epsilon}(\vec{\delta})|| = 0$ when $||\vec{\delta}|| = 0$.

The notation $||\vec{\delta}||$ represents the magnitude of the vector $\vec{\delta}$. According to this definition, a function in an error model (error function) must be zero if there are no differences between the actual and the ideal. We also expect that the magnitude of an error model will generally increase as the difference from ideal grows larger, but we have found it difficult to formalize this requirement without being overly restrictive.

One trivial error model is the set of projection functions $\pi_i(\vec{\delta}) = \delta_i$; these are zero when $||\vec{\delta}|| = 0$ and increase as the respective component of $\vec{\delta}$ increases. But the value of our error model definition is that it allows us to construct a much simpler error space with the type of QoS characteristics commonly discussed in the QoS management literature. A point in this simplified error space represents an equivalence class of output traces that are the same distance from the ideal with respect to the error model.

To continue the packet service example, if we define the service as consisting of all `deliver(packet)` messages, then both the input trace and its associated ideal output trace may contain an unbounded number of events. To define an error model for this service we need to decide how to interpret packet delivery events that are expected but never happen, or that cannot be distinguished from other packet delivery events. For this error model, we consider a packet to be lost if either the time difference in $\vec{\delta}$ for its deliver event exceeds some limit, such as 20 seconds, or its value difference is non-zero. We associate a packet delivery event with an ideal delivery event if the the actual event happened after the ideal event and there is no other interpretation that offers fewer packet losses.

To allow us to model quality at time t during the service, we define $i(p, t, \vec{\delta})$ to be the sequence of consecutive values

in $\vec{\delta}$ associated with ideal events in the interval $(t-p, t]$. Let $d(p, t, \vec{\delta})$ be the values in $i(p, t, \vec{\delta})$ associated with packets that were correctly delivered (not lost). A value of 10 seconds was chosen arbitrarily for the period p in this example.

We can then construct an error model with the following functions:

- $delay(t)$ is the mean of time differences in $d(10, t, \vec{\delta})$.
- $jitter(t)$ is the variance of time differences in $d(10, t, \vec{\delta})$.
- $loss(t)$ is the ratio of the number of packets lost in $i(10, t, \vec{\delta})$ to the number sent, or zero if none were sent.

These satisfy our definition of an error model, since each is zero when the difference values in $\vec{\delta}$ are zero and none decrease when a difference value in $\vec{\delta}$ increases.

Note that these definitions model only the error in the recent history of time t. To constrain error over the lifetime of a service we would need to use expressions like: for all time t, $delay(t) < 5$. We could define many other similar error models with different parameters or different functions. For example, it may be useful to model the 95th percentile of delay values. Still, the simple error model above is quite useful for specifying and measuring the quality of a packet delivery service and is similar to other common definitions of packet service QoS characteristics.

A point in this error space; say $delay(t) = 1$ second, $jitter(t) = 0.5$ second, and $loss(t) = 0.2$; corresponds to all output traces in which the mean delay for recent events before time t is one second, and so forth. This set of output traces forms a surface in the behavioral space that surrounds a neighborhood of the ideal output trace. Inside this neighborhood, all output traces are closer to the ideal and can be considered *better* at time t according to this error model.

We can use an error model to both quantify the *loss of quality* and to constrain it. Let $\epsilon(\vec{A} - \vec{I})$ be the tuple of error values for an error model ϵ, an actual trace A and the ideal trace for a service I. Let $limits$ be a tuple of positive real numbers representing an upper bound for each of the functions in ϵ. Then we say a service with output trace A is acceptable if for each i, $|\epsilon_i(\vec{A} - \vec{I})| < limits_i$.

Since many error models can be defined for a given service, we would like some criteria to judge which error models are better than others. The QQMM allows us to formally define desirable properties of a good error model. For brevity, we suggest only informal definitions here. We say an error model is *sound* if any set of non-zero error limits can be satisfied by some set of actual output traces. An error model is *complete* if, for any output trace that is different from the ideal, we can find a set of non-zero error limits that would exclude this trace. An error model is *minimal* if no function can be removed without losing the ability to distinguish between some output traces. We say an error model M is more *expressive* than an error model N if it can define exactly the same sets of acceptable output traces as N, and then some more.

The example error model for the packet service appears to be sound, because any non-zero limits can be satisfied by output traces with non-zero delay, jitter, and loss. The error model also appears to be complete: for any time difference value $x > 0$ in some $i(10, t, \vec{\delta})$ with n successfully delivered packets, $x/(n+1)$ is a non-zero delay limit that must be less than $|delay(t)|$ even if all $n-1$ other time differences are zero, and if $v > 0$ is a value difference in such an interval with m packets, $1/(m+1)$ is a non-zero loss limit that must be less than $|loss(t)|$ even if all other packets are delivered in a timely manner without corruption.

The error model is also minimal, as each function models an independent facet of error. The error model could be made more expressive by adding functions, to distinguish packet corruption from loss for example.

3. APPLICATION TO AN OBJECT TRACKING SERVICE

In this section, we first define error models for a real application and its component services, and then show how error for the application service can be predicted from error in the component services. We apply QQMM to an example of a class of applications we refer to as real-time content-based video analysis. These applications must process the video data at least as fast as the video data is made available and perform analysis with acceptable accuracy.

Real-time content analysis is an active research field where efficient techniques have been found for problems such as multi-object detection and tracking. In such applications, pattern classification systems which automatically classify media content in terms of high-level concepts have been adopted. Such pattern classification systems must bridge the gap between the low-level signal processing services (filtering and feature extraction) and the high-level services desired by the end-user.

The design of such a pattern classification system must select analysis algorithms that balance requirements of accuracy against requirements of timeliness.

3.1 The object tracking service

In this example, we refer to a simple object tracking service as an `Object-Tracker`. The service is simple in the sense that it can track a single object on a stationary background (i.e. the camera remains still). This example is adapted from our earlier work on supporting quality constraints in real-time content-based video analysis [12].

Figure 3 shows the `Object-Tracker` as a component receiving input from a `VideoSrc` and sending output to an `EventSink`. The `Object-Tracker` is implemented as a composition of `Feature extractor` and `Classifier` components with multicast input to the feature extractors and publish/subscribe middleware for communications between the feature extractors and the classifier. The extractors and the classifier are further decomposed into functional and communication components.

The `VideoSrc` sends each frame of an uncompressed video stream to a filter component (not shown) that divides the frame into regions, publishing the data for each to a multicast channel for that region. The multicast channel is an efficient mechanism to communicate high-bandwidth data to multiple clients over a local area network. However, in this example, exactly one `Feature Extractor` receives the data for each frame region.

Each `Feature Extractor` calculates features from the set of frame data it has received. In this example, the `Feature Extractor` components compute a two dimensional array of motion vectors; one for each block of the frame region, where a block is a subdivision of the frame into fixed size rectangles of pixels. A motion vector indicates the direction

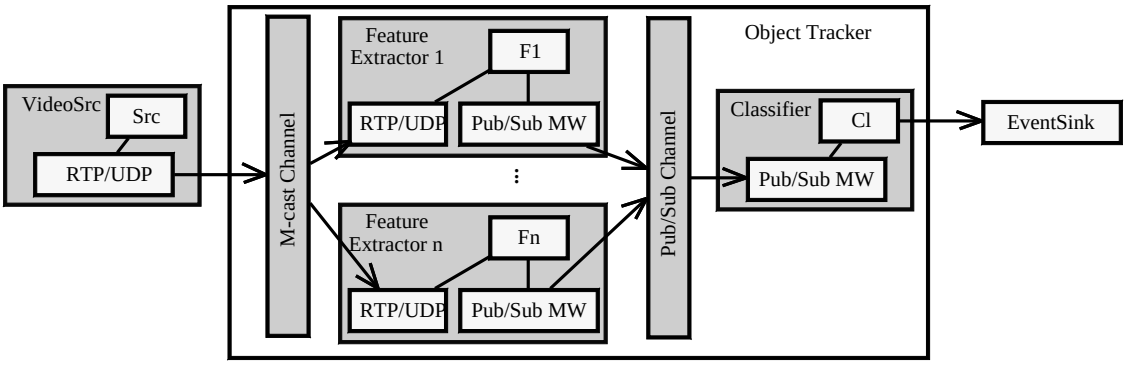

Figure 3: Functional decomposition of the `Object-Tracker`.

and distance that most pixels within a block appear to have moved from a previous frame.

The motion vectors and the associated frame number are then published by the `Feature Extractor` as event notifications on a channel for the frame region. The `Classifier` subscribes to these channels to receive motion vector data from multiple `Feature Extractor` components. In this example, the `Classifier` examines the history of motion vectors over several frames to identify and determine the center of a moving object.

We use Dynamic Bayesian Networks (DBNs) as a classifier specification language. DBNs [5] represent a particularly flexible class of pattern classifiers that allows statistical inference and learning to be combined with domain knowledge.

In order to automatically associate high-level concepts (e.g. object position) with the features produced through feature extraction, a DBN can be trained on manually annotated media streams. The goal of the training is to find a mapping between the feature space and high-level concept space, within a hypothesis space of possible mappings. After training the DBN can be evaluated by measuring the number of misclassifications on a manually annotated media stream not used in the training. This measured error rate can be used as an estimate of how accurately the DBN will classify novel media streams and can be associated with the classifier as meta data.

One important reason for using DBNs is the fact that DBNs allow features to be missing during classification at the cost of some decrease in accuracy. This fact is exploited in our work with the `Object-Tracker` to trade accuracy for timeliness as discussed below.

3.2 QoS role in planning service configuration

Even this simple `Object-Tracker` implementation requires many deployment configuration choices with associated QoS tradeoffs. Components may be deployed to different processors to work in parallel, increasing throughput and reducing delay. Throughput may also be increased by deploying pipeline components on different processors. But distribution may also increase communication overhead and add to delay.

Since the motion vector `Feature Extractor` operates locally on image regions, the performance can scale with the size of video processing task by distributing the work among a greater number of `Feature Extractor` components, each on its own processor. Similarly, the `Classifier` could be parallelized if classification should become a processing bottleneck [12]. In this paper, we consider only the case of parallelizing the `Feature Extractor`.

The amount of processing required for acceptable accuracy in object tracking is another tradeoff point. The level of accuracy provided by the `Object-Tracker` depends on the misclassification behavior (error rate) of the classifier algorithm, which in turn depends on the quality of the feature extraction input. Greater accuracy can be achieved by processing video data at a higher frame rate or higher resolution, but the increased processing requirements might increased delay in reporting the object location. Hence the configuration must be carefully selected based on both the desired level of accuracy and the tolerance for delay.

To reason about which configurations might satisfy the `Object-Tracker` QoS requirements, it must be possible to estimate what QoS the components will offer and how these QoS offers compose to satisfy the end-to-end requirements. To do this in practice, we exploit knowledge of the measured behavior of the components in the target physical processing environment.

3.3 Error model definitions

We model the `Object-Tracker` as a service that has uncompressed video frames as input and that produces a location event for every video frame as its output. A video frame number accompanies each frame, frame region, set of motion vectors, and each location output event so that there is no ambiguity about which frame these events are to be associated with. Each video frame is divided into $m \times n$ blocks. A location is represented as a block number in the video frame and indicates the center position of the tracked object.

For this service, the variables in the output trace are the time of the output and the location coordinates. It matters little whether the difference between actual and ideal location coordinates is in one axis or another, so we will refer to location as an aggregate value.

We would like to model three quality characteristics for this service: *latency*, *errorRate* and *period*. Let $i(p, t, \vec{\delta})$ be the values from the difference vector for the interval of period p up to time t as defined in the last section.

We can define the error model as follows:

- $latency(t)$ is the mean of time differences in $i(10, t, \vec{\delta})$.
- $errorRate(t)$ is the ratio of non-zero location differ-

ences in $i(10, t, \vec{\delta})$ to the total number of location values.

- *period*(t) is the maximum q such that location difference is non-zero for all values in $i(q, t', \vec{\delta})$, where $t - 10 \leq t' \leq t$.

The *latency*(t) is the mean of recent values for the elapsed time from when a frame containing a motion event is sent to the `Object-Tracker` until a causally related location event is output. The ideal latency is zero, but any real application will permit some latency greater than zero.

The *errorRate*(t) gives the fraction of the recent location difference values that were non-zero. The ideal error rate of zero may be achieved if all video blocks are processed and the moving object is similar to those used in classifier training.

The *period*(t) is the amount of time that may elapse before a updated and correct object location is reported. As with the *errorRate*, the ideal value of zero may be achieved with sufficient processing resources and good video input, but frame dropping will cause this error to increase.

Because the `Classifier` has the same output interface as the `Object-Tracker`, we can and should use the same error model. This shows a good feature of the QQMM: Error model definitions refer only to the output interface of a service and so an error model may be used for any service with the same output interface.

We model a `Feature Extractor` as a service that receives uncompressed video frames (or some region of a frame) as input and that produces a motion vector array and associated frame number as its output. The variables between actual and ideal output traces for this service are the time of output events and the motion vector array values.

For this example, we are not concerned with trading accuracy in the motion vectors against other quality dimensions so we again treat this complex data type as a single aggregate value in the following error model:

Let $i(p, t, \vec{\delta})$ be as defined in the last section.

- *latency*(t) is the mean time difference in $i(10, t, \vec{\delta})$.

- *errorRate*(t) is the ratio of non-zero motion vector array differences in $i(10, t, \vec{\delta})$ to the total number of motion vector array values.

These are the same definitions given for the functions of the same name in the previous error model except that here we reference the motion vector array as the output trace variable. Because we do not anticipate error in the deterministic algorithm for computing motion vectors, it might seem that *errorRate*(t) could be left out of our model, but this would leave our model incomplete and unable to express even the constraint that the motion vectors should be correct.

Our work with this application has been in a local area network environment where the remote communication has not been a bottleneck, but we understand that modeling the QoS of these communication links is necessary for future work. At this time, we have not defined error models for the the communication protocols or the component which divides the video frames into regions.

3.4 Modeling compositional QoS relations

To configure the `Object-Tracker` to perform with acceptable *errorRate*, *latency* and *period* requires an ability to predict these values for alternative configurations. Systems engineers accomplish this task with a combination of analysis and experimental observations of component behavior. The QQMM allows us to define an error model for each component service type that does not depend on its implementation, and thus can be used as a standard QoS specification model used by both component clients and independent component developers. Given such standards, a component developer can encode knowledge of the relation between component and composition QoS independently for each implementation. Thus, the QQMM enables a kind of QoS composition algebra: the algebraic operators are error prediction functions provided by the developer with each component implementation and the operands are the subcomponent QoS predictions.

As mentioned earlier, meta data measuring the *errorRate* for various configurations may be associated with the `Classifier`. For all of the components, measurements of processing time for a periodic task on a particular class of CPU and with a particular class of workload may be associated with the component type as meta data for use in estimating *latency*.

Another input to the *latency* prediction function is a model of the available computing resources in terms of both the number and class of CPUs, and the latency and bandwidth of communication between each pair of CPUs.

To simplify the estimation of *latency* for a service configuration, we assume that the communication between each pair of CPUs is contention free and that the latency and bandwidth are constant. These assumptions are valid for a significant class of distributed processing environments (e.g. dedicated homogeneous computers connected in a dedicated switched LAN) and allow us to ignore the complexity of communication contention, routing, etc., which are not the focus of this paper.

Given an allocation of components to processors, the processing period of each `Object-Tracker` component can be estimated. From the component QoS predictions and configuration values such as the classifier location output rate, the end-to-end error for this example of the `Object-Tracker` can be predicted.

The composition of *latency* for serial tasks, such as remote communication of video frame data and `Feature Extractor` processing of that frame, is the sum of the delays. The QQMM semantics ensure that this composition is seamless: that end-to-end accounting of time attributes every moment to exactly one service in the sequence. If the latency measurements follow these semantics than there is good reason to hope that the composition estimate will be good.

The serial composition of `Feature Extractor` and `Classifier` is not strictly serial processing however. The `Classifier` does not wait for all features from a frame to arrive before reporting a guess about the location, but instead is configured with its own periodic schedule to update hypotheses, including hypotheses about past. In this implementation, the composition trades the risk of an increase in the *errorRate* to avoid the high *latency* of waiting for every serial dependency. The end-to-end *latency* is held constant at runtime while the *errorRate* may vary.

The prediction of the *errorRate* can be made analyti-

cally from estimates of the availability of extracted features and knowledge of the classifier. The ARCAMIDE algorithm exploits the fact that DBNs allow features to be missing during classification to trade accuracy for timeliness [12]. Alternatives are generated for removing feature extractors from an initial configuration. The error prediction for the `Object-Tracker` is used to sort the alternatives, least increase in the *errorRate* first. This algorithm then tests each configuration to determine if the *latency* and *period* prediction will satisfy application requirements. In this sequence, the first service configuration which meets the *latency* and *period* requirements, must also meet the accuracy requirement, otherwise it is not possible to satisfy the requirements with the specified processing resources.

The *period* can be predicted from the absolute value of the difference between the classifier update period and the $VideoSrc$ frame period.

This example suggests that the error prediction for a composition can be a complex function of component interactions and component error. The QQMM semantics enable such error prediction functions to be written without knowledge of component implementations and thus, to predict error regardless of which component implementation is plugged in to the composition. However, we have only begun to define error models for real services and error predictions for compositions. Future work is needed to learn if there are important patterns for error prediction in compositions.

4. RELATED WORK

There has been little work published specifically addressing QoS models for component architectures. Researchers at BBN developed QuO (Quality of Service for Objects) [13] as a framework for management of QoS properties in CORBA applications. A more recent QuO paper introduced *qoskets* as a means for reusing adaptive QoS behaviors, but they have not yet adapted this work to a component architecture [10]. They specify QoS contracts between client and server objects, but do not address other service types such as a pipeline component that receives input from one object and sends output to another. Also, they do not focus on QoS semantics, so it is not clear that a description of QoS provided by one component would be understood correctly by an independent developer who would use such a component.

SLAng is a language for specifying service level agreements between very large grain components that may be owned and operated by separate commercial entities [3]. The authors acknowledge that SLAng's informal semantics for QoS characteristics is a weakness [2]. Our model is complementary and could provide a formal semantics and a method to expand the SLAng catalog with QoS characteristics for new service types.

Nahrstedt, et al., describe a QoS-aware middleware architecture designed to support QoS-sensitive applications [7]. They propose a QoS compiler to map from user-perceived QoS down to system-level QoS, but we understand that this compiler understands only a fixed set of QoS characteristics. They conclude that more research is needed to allow uniform specification of QoS for different application domains. We agree and we believe that QQMM provides a prerequisite common semantics independent of any particular middleware or component architecture.

5. CONCLUSIONS

In this paper we have described the QuA QoS Meta-Model (QQMM) for defining QoS semantics for an arbitrary service. The key features of the QQMM are a definition of a service that is based on component interface semantics and a definition of quality dimensions based on a metric space with ideal service behavior at the origin. Error models that conform to the QQMM define QoS dimensions that can be observed by a client without knowledge of the service implementation. This allows independent component developers to agree on a common standard for specifying client QoS requirements and component QoS offers. The QQMM allows us to analyze any implementation as a composition of component services and to define error models for these component services that fully account for loss of quality in the implementation. From this analysis, a component developer may be able to provide a mapping relation between QoS of component services and the QoS of a composition.

The QQMM provides clear guidelines for defining QoS measures with a rigorous semantics, but we have learned from initial experiments that it can be difficult to define precise error models that correspond to our intuition about quality dimensions. Despite this note of caution, we believe a strong semantics are a prerequisite for QoS management in component-base software engineering where representations of QoS properties come with "off-the-shelf" components.

5.1 Acknowledgment

Thanks to Jonathan Walpole for comments on an early draft and to the referees for their suggestions to improve the paper. This work was funded in part by a grant from the Research Council of Norway.

6. REFERENCES

[1] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *ACM Distributed Computing Journal*, 15(2):109–126, 2002.

[2] D. Davide Lamanna and James Skene and Wolfgang Emmerich. Specification Language for Service Level Agreements, 2003. http://www.newcastle.research.ec-.org/tapas/deliverables/D2.pdf.

[3] Wolfgang Emmerich, D. Davide Lamanna, Giacomo Piccinelli, and James Skene. Method for service composition and analysis, 2003. http://www.newcastle.research.ec.org/tapas/deliverables/d3.pdf.

[4] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002.

[5] F. V. Jensen. Bayesian networks and decision graphs, 2001. Series for Statistics and Engineering and Information Science, Springer Verlag.

[6] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, pages 20–22, Kyoto, Japan, 1998.

[7] Klara Nahrstedt, Dongyan Xu, Duangdao Wichadakul, and Baochun Li. QoS-aware middleware for ubiquitous computing. *IEEE Communications*

Magazine, 39(11):140–148, November 2001.

[8] I. Pyarali, D. Schmidt, and R. Cytron. Achieving end-to-end predictability of the TAO real-time CORBA ORB. In *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, San Jose, CA, 2002.

[9] Richard Staehli, Frank Eliassen. Component-Based Service Planning For Platform-Managed QoS. Submitted to Middleware 2004, 2004.

[10] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal. Packaging quality of service control behaviors for reuse. In *Proceedings of ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing*, Washington, DC, 2002.

[11] Richard Staehli and Jonathan Walpole. Quality of service specifications for multimedia presentations. *Multimedia Systems*, 3(5/6), 1995.

[12] Viktor S. Vold Eide and Frank Eliassen and Ole-Christoffer Granmo and Olav Lysne. Supporting Timeliness and Accuracy in Distributed Real-Time Content-based Video Analysis. In *ACM Multimedia*, 2003.

[13] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3, 1997.