

## **Quality-of-Service Specification in Distributed Object Systems**

Svend Frølund, Jari Koistinen  
Software Technology Laboratory  
HPL-98-159  
September, 1998

E-mail: [frolund,jari]@hpl.hp.com

quality-of-service  
specification,  
distributed object  
systems,  
software design,  
quality-of-service-  
enabled systems

Traditional object-oriented design methods deal with the functional aspects of systems, but they do not address quality of service (QoS) aspects, such as reliability, availability, performance, security, and timing. However, deciding which QoS properties should be provided by individual system components is an important part of the design process. Different decisions are likely to result in different component implementations and system structures. Thus, decisions about component-level QoS should commonly be made at design time, before the implementation is begun. Since these decisions are an important part of the design process, they should be captured as part of the design. We propose a general Quality-of-Service specification language, which we call QML. In this paper we show how QML can be used to capture QoS properties as part of designs. In addition, we extend UML, the de-facto standard object-oriented modeling language, to support the concepts of QML.

QML is designed to integrate with object-oriented features, such as interfaces, classes, and inheritance. In particular, it allows specification of QoS properties through refinement of existing QoS specifications. Although we exemplify the use of QML to specify QoS properties within the categories of reliability and performance, QML can be used for specification within any QoS category— QoS categories are user-defined types in QML.

Sometimes, QoS characteristics and requirements change dynamically due to changing user preferences, or changes in the environment. For such situations static specification is insufficient. To allow for dynamic systems that change and evolve over time, we provide a QoS specification run-time representation. This representation enables systems to create, manipulate, and exchange QoS information, and thereby negotiate and adapt to changing QoS requirements and conditions.

## 1. Introduction

### 1.1. Quality-of-Service in Software Design

In software engineering—like any engineering discipline—design is the activity that allows engineers to invent a solution to a problem. The input to the design activity consists of various requirements and constraints. The result of a design activity is a solution in which all major architectural and technical problems have been addressed. Design is an important activity since it allows engineers to invent solutions stepwise and in an organized manner. It makes engineers consider solutions and trade various system functions against each other.

To be useful, computer systems must deliver a certain *quality of service* (QoS) to its users. By QoS, we refer to non-functional properties such as performance, reliability, availability, and security. Although the delivered QoS is an essential aspect of a computer system, traditional design methods, such as [5, 22, 4, 12, 7], do not incorporate QoS considerations into the design process. We strongly believe that, in order to build systems that deliver their intended QoS, it is essential to systematically take QoS into account at design time, and not as an afterthought during implementation.

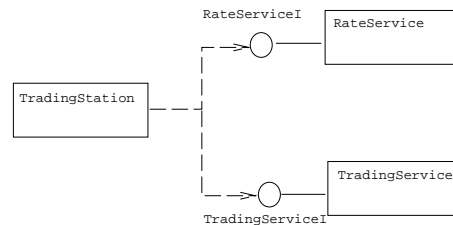


Figure 1. Class diagram for the currency trading system

We use a simple example to illustrate the need for design-time QoS considerations. Consider the currency trading system in Figure 1. Currency traders interact with the **trading station**, which provides a user interface. To provide its functionality, the **trading station** uses a **rate service** and a **trading service**. The **rate service** provides rates, interests, and other information important to foreign exchange trading. The **trading service** provides the mechanism for making trades in a secure way. An inaccessible currency trading system might incur significant financial loss, therefore it is essential that the system is highly available.

It is important, at design time, to decide the QoS properties of individual system components. For example, we need to decide the availability properties of the **rate service**. We can decide that the **rate service** should be highly available so that the **trading station** can rely exclusively on it for rate information. Alternatively, we can decide that the **rate service** need *not* be highly available. If the **rate service** is not highly available, the **trading station** cannot rely exclusively on it, but must be prepared to continue operation if the **rate service** fails. To continue operation, the **trading station** could connect to an external rate service. As the example shows, different availability properties for the **rate service** can result in different system architectures. It is important to decide on particular QoS properties, and thereby chose a specific architecture, at design time.

```

interface RateServiceI {
  Rates latest(in Currency c1,in Currency c2) raises (InvalidC);
  Forecast analysis(in Currency c) raises (Failed);
};

```

**Figure 2.** The RateServiceI interface

Besides the system architecture, the choice of QoS properties for individual components also affects the implementation of components. For example, the `rate service` can be implemented as a single process or as a process pair, where the process-pair implementation provides higher availability. Different QoS properties are likely to require different implementations. Moreover, the QoS properties of a component may affect the implementation of its clients. For example, with a single-process implementation, the `trading station` may have to explicitly detect failures and restart the `rate service`, whereas with a process-pair implementation, failures may be completely masked for the `trading station`.

### 1.2. Quality-of-Service Specification

In the previous section we argued that QoS properties of individual components reflect important design decisions, and that we need describe these QoS properties as part of the design process. To capture component-level QoS properties, we introduce a language called QML (QoS Modeling Language).

Consider the CORBA IDL [17] interface definition for the `rate service` in Figure 2. A rate service provides one operation for retrieving the latest exchange rates with respect to two currencies. The other operation performs an analysis and returns a forecast for the specified currency. The interface definition specifies the syntactic signature for a service but does not specify any semantics or non-functional aspects. In contrast, we concern ourselves with how to specify the required or provided QoS for servers implementing this interface.

QML has three main abstraction mechanisms for QoS specification: *contract type*, *contract*, and *profile*. QML allows us to define contract types that represent specific QoS aspects, such as performance or reliability. A contract type defines the *dimensions* that can be used to characterize a particular QoS aspect. A dimension has a domain of values that may be ordered. There are three kinds of domains: *set* domains, *enumerated* domains, and *numeric* domains. A contract is an instance of a contract type and represents a particular QoS specification. Finally, QML profiles associate contracts with interfaces, operations, operation arguments, and operation results.

The QML definitions in Figure 3 include two contract types `Reliability` and `Performance`. The reliability contract type defines three dimensions. The first one represents the number of failures per year. The keyword “decreasing” indicates that a smaller number of failures is better than a larger one. Time-to-repair (TTR) represents the time it takes to repair a service that has failed. Again, smaller values are better than larger ones. Finally, `availability` represents the probability that a service is available. In this case, larger values represent stronger constraints while smaller values represent lower probabilities and are therefore weaker.

We also define a contract named `systemReliability` of type `Reliability`. The

```

type Reliability = contract {
  numberOfFailures: decreasing numeric no/year;
  TTR: decreasing numeric sec;
  availability: increasing numeric;
};

type Performance = contract {
  delay: decreasing numeric msec;
  throughput: increasing numeric mb/sec;
};

systemReliability = Reliability contract {
  numberOfFailures < 10 no/year;
  TTR {
    percentile 100 < 2000;
    mean < 500;
    variance < 0.3
  };
  availability > 0.8;
};

rateServerProfile for RateServiceI = profile {
  require systemReliability;
  from latest require Performance contract {
    delay {
      percentile 50 < 10 msec;
      percentile 80 < 20 msec;
      percentile 100 < 40 msec;
      mean < 15 msec
    };
  };

  from analysis require Performance contract {
    delay < 4000 msec
  };
};

```

Figure 3. Contracts and Profile for RateServiceI

contract specifies constraints that can be associated with, for example, an operation. Since the contract is named it can be used in more than one profile. In this case, the contract specifies an upper bound on the allowed number of failures. It also specifies an upper bound, a mean, and a variance for TTR. Finally, it states that availability must always be greater than 0.8.

Next we introduce a profile called `rateServerProfile` that associates contracts with operations in the `RateServiceI` interface. The first requirement clause states that the server should satisfy the previously defined `systemReliability` contract. Since this requirement is not related to any particular operation, it is considered a default requirement and holds for every operation. Contracts for individual operations are allowed only to strengthen (refine) the default contract. In this profile there is no

default performance contract; instead we associate individual performance contracts with the two operations of the `RateServiceI` interface. For `latest` we specify in detail the distribution of delays in percentiles, as well as an upper bound on the mean delay. For `analysis` we specify only an upper bound and can therefore use a slightly simpler syntactic construction for the expression. Since throughput is omitted for both operations, there are no requirements or guarantees with respect to this dimension.

We have now effectively specified reliability and performance requirements on any implementation of the `rateServiceI` interface. The specification is syntactically separate from the interface definition, allowing different `rateServiceI` servers to have different QoS characteristics.

QoS specifications can be used in many different situations. They can be used during the design of a system to understand the QoS requirements for individual components that enable the system as a whole to meet its QoS goals. Such design-time specification is the focus of this paper. QoS specifications can also be used to dynamically negotiate QoS agreements between clients and servers in distributed systems.

In negotiation it is essential that we can match offered and required QoS characteristics. As an example, satisfying the constraint “delay < 10 msec” implies that we also satisfy “delay < 20 msec.” We want to enable automatic checking of such relations between any two QoS specifications. We call this procedure *conformance checking*, and it is supported by QML.

QML allows designers to specify QoS properties independently of how these properties can be implemented. For example, QML enables designers to specify a certain level of availability without reference to a particular high-availability mechanism such as primary-backup or active replication.

QML supports the specification of QoS properties in an object-oriented manner; it provides abstraction mechanisms that integrate with the usual object-oriented abstraction mechanisms such as classes, interfaces, and inheritance. Although QML is not tied to any particular design notation, we show how to integrate QML with UML [5], and we provide a graphical syntax for component-level QoS specifications.

QML is a general-purpose QoS specification language; it is not tied to any particular domain, such as real-time or multi-media systems, or to any particular QoS *category*, such as reliability or performance.

### 1.3. Runtime Usage of QoS Specifications

Static QoS specifications are valuable, but have their limitations. they do not allow objects of a system to be aware of the QoS they require or provide. In contrast, an object that is QoS-aware is able to communicate and manipulate its QoS information. It is also able to compare its QoS needs with what is provided by other objects.

To support QoS awareness, our goal is to allow the usage of QoS specifications as first-class runtime entities. To facilitate efficient description of QoS specifications, such runtime entities should be based on the same concepts as QML. We have designed a QML-based runtime QoS fabric, which we call QRR (QoS Runtime Representation).

QRR allows dynamic creation, manipulation, communication and comparison of QoS specifications. Since QRR has the same concepts as QML, we can provide efficient translations between the two representations. Moreover, the precise definition of QML’s semantics carries over to QRR.

QRR enables the construction of QoS-aware systems that can select services

dynamically based on their QoS capabilities. It also allows objects to provide different levels of QoS based on current conditions, and communicate their current level of QoS to other objects.

We organize the rest of this paper in the following way. In Section 2, we introduce our terminology for distributed object systems. We present the dimensions of reliability and performance that we use in Section 3. We describe QML in Section 4, and we explain its integration into UML in Section 5. We use QML and the UML extensions to specify the QoS properties of a computer-based telephony system in Section 6. In Section 7 we describe a runtime representation for QoS specification information. The topic of Section 8 is related work, and Section 9 is a discussion of our approach. Finally, in Section 10, we draw our conclusions.

## 2. Our Terminology for Object-Oriented Systems

We assume that a system consists of a number of *services*. A service has a number of *clients* that rely on the service to get their work done. A client may itself provide service to other clients.

A service has a *service specification* and an implementation. A service specification describes what a service provides; a service implementation consists of a collection of software and hardware objects that collectively provide the specified service. For example, a name service maintains associations between names and objects. A name service can be replicated, that is, it can be implemented by a number of objects that each contain all the associations. It is important to notice that we consider a replicated name service as *one* logical entity even though it may be implemented by a collection of distributed objects.

A client uses a service through a *service reference*, or simply a *reference*. A reference is a handle that a client can use to issue service requests. A reference provides a client with a single access point, even to services that are implemented by multiple objects.

Traditionally, a service specification is a functional *interface* that lists the operations and attributes that clients can access; we extend this traditional notion of a service specification to also include a definition of the QoS provided by the service. The same service specification can be realized by multiple implementations, and the same collection of objects can implement multiple service specifications.

## 3. Selected Dimensions

To specify QoS properties in QML, we need a way to formally quantify the various aspects of QoS. A *QoS category* denotes a specific non-functional characteristic of systems that we are interested in specifying. *Reliability*, *security*, and *performance* are examples of such categories. Each category consists of one or more *dimensions* that represent a metric for one aspect of the category. *Throughput* would be a dimension of the performance QoS category. We represent QoS categories and dimensions as user-defined types in QML.

To meaningfully characterize services with QoS categories we need valid dimensions. We are particularly interested in the dimensions that characterize services without exposing internal design and implementation details. Such dimensions enable the specification of QoS properties that are relevant and understandable for, in principle, any service regardless of implementation technology.

We describe a set of dimensions for reliability and performance. In [13] we have reviewed a variety of literature and systems on reliability including work by Gray et al. [11], Cristian [8], Reibman [19], Birman, [3], Maffeis [16], Littlewood [15], and others. As a result we propose the dimensions in Figure 4 for characterizing the reliability of distributed object systems.

Name	Type
TTR	Time
TTF	Time
Availability	Probability
Continuous availability	Probability
Failure masking	<b>set</b> {failure, omission, response, value, state, timing, late, early}
Server failure	<b>enum</b> {halt, initialState, rollBack}
Operation semantics	<b>enum</b> {exactlyOnce, atLeastOnce, atMostOnce}
Rebinding policy	<b>enum</b> {rebind, noRebind}
Number of failures	Unsigned Integer
Data policy	<b>enum</b> {valid, notValid}

Figure 4. Reliability dimensions

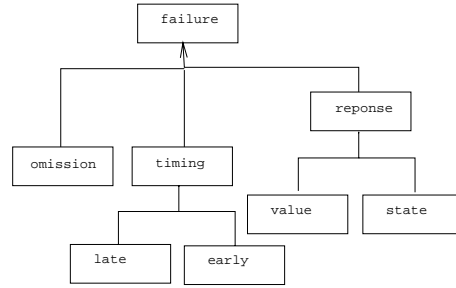
We use the measurable quantities of time to failure (TTF) and time to repair (TTR). **Availability** is the probability that a service is available when a client attempts to use it. Assume for example that service is down totally one week a year, then the availability would be 51/52, which is approximately 0.98. **Continuous availability** assesses the probability with which a client can access a service an infinite number of times during a particular time period. The service is expected not to fail and to retain all state information during this time period. We could for example require that a particular client can use a service for a 60 minute period without failure with a probability of 0.999. **Continuous availability** is different from **availability** in that it requires subsequent use of a service to succeed but only for a limited time period.

The **failure masking** dimension is used to describe what kind of failures a server may expose to its clients. A client must be able to detect and handle any kind of exposed failure. The above table lists the set of all possible failures that can be exposed by services in general. The QoS specification for a particular service will list the subset of failures exposed by that service.

We base our categorization of failure types—shown in Figure 5—on the work by Cristian [8]. If a service exposes **omission** failures, clients must be prepared to handle a situation where the service simply omits to respond to requests. If a service exposes **response** failures, it might respond with a faulty return value or an incorrect state transition. Finally, if the service exposes **timing** failures, it may respond in an untimely manner. Timing failures have two subtypes: **late** and **early** timing errors. Services can have any combination of failure masking characteristics.

**Operation semantics** describe how requests are handled in the case of a failure. We can specify that issued requests are executed **exactlyOnce**, **atLeastOnce**, or **atMostOnce**.

**Server failure** describes the way in which a service can fail. That is, whether it will **halt** indefinitely, restart in a well defined **initialState**, or restart **rolledBack** to a previous check point.



**Figure 5.** Failure type hierarchy

The number of failures gives a likely upper bound for the number of times the service will fail during a specific time period.

When a service fails the client needs to know whether it can use the existing reference or whether it needs to rebind to the service after the service has recovered. The **rebinding policy** is used to specify this aspect of reliability.

Finally, we propose that the client also needs to know if data returned by the service still is valid after the service has failed and been restarted. To specify this we need to associate **data policy** with entities such as return values and out arguments.

For the purpose of this paper we will propose a minimal set of dimensions for characterizing performance. We are only including **throughput** and **latency**. **Throughput** is the transfer rate for information, and can, for example, be specified as megabytes per second. **Latency** measures the time between the point that an invocation was issued and the time at which the response was received by the client.

Dimensions such as those presented here constitute the vocabulary for QoS specification languages. We use the dimensions to describe the example in section 6.

#### 4. QML: A Language to Specify QoS Properties

We describe the main design considerations for QML in Section 4.1. We already introduced the fundamental concepts of QML in section 1. Sections 4.2–4.8 describe the syntax and semantics of QML in more detail. For the full description of QML we refer to the language definition in [9].

##### 4.1. Basic Requirements

The main design consideration for QML is to support QoS specification in an object-oriented context. We want QML to integrate seamlessly with existing object-oriented concepts. This overall goal results in the following specific design requirements for QML:

- QoS specifications should be syntactically separate from other parts of service specifications, such as interface definitions. This separation allows us to specify different QoS properties for different implementations of the same interface.
- It should be possible to specify both the QoS properties that clients require and the QoS properties that services provide. Moreover, these two aspects should be specified separately so that a client-server relationship has two QoS specifications:



a specification that captures the client’s requirements and a specification that captures the service’s provisioning. This separation allow us to specify the QoS characteristics of a component, the QoS properties that it provides and requires, without specifying the interconnection of components. The separation is essential if we want to specify the QoS characteristics of components that are reused in many different contexts.

- There should be a way to determine whether the QoS specification for a service satisfies the QoS requirement of a client. This requirement is a consequence of the separate specification of the QoS properties that clients require and the QoS properties that services provide.
- QML should support refinement of QoS specifications. In distributed object systems, interface definitions are typically subject to inheritance. Since inheritance allows an interface to be defined as a refinement of another interface, and since we associate QoS specifications with interfaces, we need to support refinement of QoS specifications.
- It should be possible to specify QoS properties at a fine-grained level. As an example, performance characteristics are commonly specified for individual operations. As another example, the data policy dimension described in Section 3 is applicable to arguments and return values of operations. QML must allow QoS specifications for interfaces, operations, attributes, operation parameters, and operation results.

Other aspects such as negotiation and utility can be dealt with as mechanisms using QML or possibly be part of future extensions of QML. This paper focuses on the requirements listed above.

We have already briefly introduced the fundamental concepts of QML: *contract type*, *contract*, *profile*. The following sections will provide a more detailed description of QML.

#### 4.2. Contracts and Contract Types

A contract type contains a *dimension type* for each of its dimensions. We use three different dimension types: set, enumeration, and numeric. Figure 6 gives an abstract syntax for contract and dimension types.

Contracts are instances of contract types. A contract type defines the structure of its instances. In general, a contract contains a list of constraints. Each constraint is associated with a dimension. For example, if we have a dimension “latency” in a contract type, a contract instance may contain the constraint “latency < 10.” Figure 7 gives an abstract syntax for contracts and constraints.

A contract may specify constraints for all or a subset of the dimensions in its contract type. Omission of a specification for a particular dimension indicates that the contract is trivially satisfied along that dimension.

In general, a constraint consists of a name, an operator, and a value. The name is typically the name of a dimension, but, as we describe in Section 4.3, the name can also be the name of a dimension *aspect*. The permissible operators and values depend on the dimension type. A dimension type specifies a domain of values. These values can be used in constraints for that dimension. The domain may be ordered. For example, a numeric domain comes with a built-in ordering (“<”) that corresponds to the usual ordering on numbers. Set and enumeration domains do not come with

<i>conType</i>	::=	<b>contract</b> { <i>dimName</i> <sub>1</sub> : <i>dimType</i> <sub>1</sub> ; ... ; <i>dimName</i> <sub>k</sub> : <i>dimType</i> <sub>k</sub> ; }
<i>dimName</i>	::=	<i>n</i>
<i>dimType</i>	::=	<i>dimSort</i>   <i>dimSort unit</i>
<i>dimSort</i>	::=	<b>enum</b> { <i>n</i> <sub>1</sub> , ... , <i>n</i> <sub>k</sub> }   <i>relSem enum</i> { <i>n</i> <sub>1</sub> , ... , <i>n</i> <sub>k</sub> } <b>with order</b>   <b>set</b> { <i>n</i> <sub>1</sub> , ... , <i>n</i> <sub>k</sub> }   <i>relSem set</i> { <i>n</i> <sub>1</sub> , ... , <i>n</i> <sub>k</sub> }   <i>relSem set</i> { <i>n</i> <sub>1</sub> , ... , <i>n</i> <sub>k</sub> } <b>with order</b>   <i>relSem numeric</i>
<i>order</i>	::=	<b>order</b> { <i>n</i> <sub>i</sub> < <i>n</i> <sub>j</sub> , ... , <i>n</i> <sub>k</sub> < <i>n</i> <sub>m</sub> }
<i>unit</i>	::=	<i>unit/unit</i>   %   <i>msec</i>   ...
<i>relSem</i>	::=	<b>decreasing</b>   <b>increasing</b>

Figure 6. Abstract syntax for contract types

<i>contract</i>	::=	<b>contract</b> { <i>constraint</i> <sub>1</sub> ; ... ; <i>constraint</i> <sub>k</sub> ; }
<i>constraint</i>	::=	<i>dimName constraintOp dimValue</i>   <i>dimName</i> { <i>aspect</i> <sub>1</sub> ; ... ; <i>aspect</i> <sub>n</sub> ; }
<i>dimValue</i>	::=	<i>literal unit</i>   <i>literal</i>
<i>literal</i>	::=	<i>n</i>   { <i>n</i> <sub>1</sub> , ... , <i>n</i> <sub>k</sub> }   <i>number</i>
<i>aspect</i>	::=	<b>percentile</b> <i>percentNum constraintOp dimValue</i>   <b>mean</b> <i>constraintOp dimValue</i>   <b>variance</b> <i>constraintOp dimValue</i>   <b>frequency</b> <i>freqRange constraintOp number%</i>
<i>freqRange</i>	::=	<i>dimValue</i>   <i>lRangeLimit dimValue</i> , <i>dimValue rRangeLimit</i>
<i>lRangeLimit</i>	::=	(   [
<i>rRangeLimit</i>	::=	)   ]
<i>constraintOp</i>	::=	==   >=   <=   <   >
<i>percentNum</i>	::=	0   1   ...   99   100
<i>dimName</i>	::=	<u>defined in Figure 6</u>
<i>unit</i>	::=	<u>defined in Figure 6</u>

Figure 7. Abstract syntax for contracts

a built-in ordering; for those types of domains we have to describe a user-defined ordering of the domain elements. The domain ordering determines which operators can be used in constraints for that domain. For example, we cannot use inequality operators (“<,” “>,” “<=,” “>=”) in conjunction with an unordered domain.

The domain for a set dimension contains elements that are sets of name literals.

```

type T = contract { // A contract type expression
  s1: decreasing set { e1, e2, e3, e4 } with order {e2<e1, e1<e3, e3<e4};
  e1: increasing enum { a1, a2, a3 } with order {a2<a1, a3<a2};
  n1: increasing numeric mb / sec;
};

T contract { // A contract expression of type T
  s1 <= { e1, e2 };
  e1 < a2;
  n1 < 23;
};

```

Figure 8. Example contract type and contract expressions

We specify a set domain using the keyword **set**, as in “**set**  $\{n_1, \dots, n_k\}$ .” This defines a set domain where the domain elements are subsets of the set “ $\{n_1, \dots, n_k\}$ .” The constraints over a set dimension will then be constraints with set values, as in “**failures** == {**response**, **omission**}.”

The domain for an enumeration dimension contains elements that are name literals. We specify an enumeration domain using the keyword **enum**. For example, we could define an enumeration domain as follows: “**enum**  $\{n_1, \dots, n_k\}$ .” Here, the domain will contain the name literals “ $n_1, \dots, n_k$ ,” and we can specify constraints as “**dataPolicy** == **valid**.”

The domain of a numeric dimension contains elements that are real numbers. Constraints for a numeric dimension are written as “**latency** < 10.”

Elements of numeric dimensions are always ordered. We can specify a user-defined ordering for set and enumerated dimensions in the following way: “**order** {**valid** < **invalid**}.” When dimensions are ordered we need to specify whether larger or smaller values are considered stronger. As an example consider the dimension of availability. A larger numeric value for availability is a stronger that a smaller, we say that availability is an “increasing” dimension. Other dimensions, such as delay, are “decreasing” since smaller values are consider as stronger guarantees. Consequently, QML requires that we define ordered dimensions as either decreasing or increasing. For the data validity enum decreasing semantics seems most intuitive, since **valid** also satisfies **invalid**.

The example in Figure 8 gives an example of a contract type expression followed by a contract expression. Note that the contract expression is explicitly typed with a contract type name, this explicit typing enables the QML compiler to determine a unique contract type for any contract expression. So far we have only covered the syntax for contract values and contract types. In Section 4.4, we describe how to name contract values and contract types, and how to use those names in contract expressions.

### 4.3. Aspects

In addition to simple constraints QML supports more complex characterizations that are called *aspects*. An aspect is a statistical characterization; QML currently includes four generally applicable aspects: *percentile*, *mean*, *variance*, and *frequency*. Aspects are used for characterizations of measured values over some time period.

```

contractTypeName contract {
  s1 { percentile 20 < { e1, e2 } };
  e1 {
    frequency a1 <= 10 %;
    frequency a2 >= 80 %;
  };
  n1 {
    percentile 10 < 20;
    percentile 50 < 45;
    percentile 90 < 85;
    percentile 100 <= 120;
    mean >= 60;
    variance < 0.6;
  };
};

```

Figure 9. Example contract expression

The percentile aspect defines an upper or lower value for a percentile of the measured entities. The statement **percentile**  $P$  denotes the strongest  $P$  percent of the measurements or occurrences that have been observed. The aspect “**percentile** 80 < 6” states that the 80th percentile of measurements for the dimensions must be less than 6. We allow a constraint for a dimension to contain more than one percentile aspect, as long as the same percentile  $P$  does not occur more than once.

QML also allows the specification of frequency constraints for individual values which is useful with enumerated types, and for ranges, which is useful with numeric dimensions. Rather than specifying specific numbers for the frequency, QML allows us to specify the relative percentage with which values in a certain range occur. The constraint “**frequency**  $V > 20\%$ ” means that in more that 20% of the occurrences we should have the value  $V$ . The literal  $V$  can be a single value or if the dimension has an ordering, and only then, it may be a range. The constraint “**frequency** [1, 3] > 35%” means that we expect 35% of the actual occurrences to be larger than 1 and less than or equal to 3.

Figure 9 shows some examples of aspects in contract expressions. The contract expression is preceded by the name of its corresponding contract type. For **s1** we define one constraint for the 20th percentile. The meaning of this is that the strongest 20% of the value must be less than the specified set value.

For **e1** we define the frequencies that we expect for various values. For the value **a1** we expect a frequency of less than or equal to 10%. For **a2** we expect a frequency greater than or equal to 80%, and so forth.

The constraint on **n1** defines bounds for values in different percentiles over the measurements of **n1**. In addition, we define an upper bound for the mean and the variance.

#### 4.4. Definition of Contracts and Contract Types

The definition of a contract type binds a name to a contract type; the definition of a contract binds a name to the value of a contract expression. Figure 10 illustrates the abstract syntax to define contracts and contract types. In the abstract syntax, we use

<code>decl</code>	<code>::= conTypeDecl   conDecl</code>
<code>conTypeDecl</code>	<code>::= type y = conType</code>
<code>conDecl</code>	<code>::= x<sub>c</sub> = conExp</code>
<code>conExp</code>	<code>::= y contract</code>
	<code>  x<sub>c</sub> refined by</code>
	<code>{constraint<sub>1</sub>; ...; constraint<sub>k</sub>};</code>
<code>conType</code>	<code>::= <u>defined in Figure 6</u></code>
<code>contract</code>	<code>::= <u>defined in Figure 7</u></code>
<code>constraint</code>	<code>::= <u>defined in Figure 7</u></code>

Figure 10. Abstract syntax for definition of contracts and contract types

```

type Reliability = contract {
  failureMasking: decreasing set {omission, lostResponse, noExecution,
    response, responseValue, stateTransition};
  serverFailure: enum {halt, initialState, rolledBack};
  operationSemantics: decreasing enum {atLeastOnce, atMostOnce, once} with
    order {once < atLeastOnce, once < atMostOnce};
  rebindingPolicy: decreasing enum {rebind, noRebind} with
    order {noRebind < rebind};
  dataPolicy: decreasing enum {valid, invalid} with
    order {valid < invalid};
  numOffFailure: decreasing numeric failures/year;
  MTTR: decreasing numeric sec;
  MTTF: increasing numeric day;
  reliability: increasing numeric;
  availability: increasing numeric;
};

```

Figure 11. Example contract type definition

$x_c$  as a generic name for contracts and  $y$  as a generic name for contract types.

We can define a contract  $B$  to be a refinement of another contract  $A$  using the construct “ $B = A$  **refined by**{...}” where  $A$  is the name of a previously defined contract. The contract that is enclosed by curly brackets ({...}) is a “delta” that describes the difference between the contracts  $A$  and  $B$ . We say that the delta *refines*  $A$  and that  $B$  is a *refinement* of  $A$ . The delta can specify QoS properties along dimensions for which specification was omitted in  $A$ . Furthermore, the delta can replace specifications in  $A$  with stronger specifications. The notion of “stronger than” is given by a conformance relation on constraints. We describe conformance in more detail in Section 4.8.

Figure 11 and Figure 12 illustrates how a named contract type (Reliability) can be define and how contracts of that type can be defined respectively. The contract type Reliability has the dimensions that we have identified within the QoS category of reliability described in section 3

The contract `systemReliability` is an instance of Reliability; it captures a system wide property, namely that operation invocation has “exactly once” (or transactional) semantics. The `systemReliability` only provides a guarantee about

```

systemReliability = Reliability contract {
  operationSemantics == once;
};

nameServerReliability = systemReliability {
  serverFailure == rolledBack;
};

type Performance = contract {
  latency: decreasing numeric msec;
  throughput: increasing numeric kb/sec;
};

traderResponse = Performance contract {
  latency { percentile 90 < 50 msec };
};

```

Figure 12. Example contract definitions

the invocation semantics, and does not provide any guarantees for the other dimensions specified in the `Reliability` contract type.

The contract `nameServerReliability` is defined as a refinement of another contract, namely the contract bound to the name `systemReliability`. In the example, we strengthen the `systemReliability` contract by providing a specification along the `serverFailure` dimension, which was left unspecified in the `systemReliability` contract.

#### 4.5. Profiles

According to our definition, a service specification contains an interface and a QoS profile. The interface describes the operations and attributes exported by a service; the profile describes the QoS properties of the service. A profile is defined relative to a specific interface, and it specifies QoS contracts for the attributes and operations described in the interface. We can define multiple profiles for the same interface, which is necessary since the same interface can for example have multiple implementations with different QoS properties.

Once defined, a profile can be used in two contexts: to specify client QoS requirements and to specify service QoS provisioning. Both contexts involve a *binding* between a profile and some other entity. In the client context this other entity is the service reference used by the client; in the service context, the entity is a service implementation. We discuss bindings in Section 4.7. Here, we describe a syntax for profile values, and in Section 4.6 we describe a syntax for profile definition.

Figure 13 gives an abstract syntax for profiles. A profile is a list of requirements, where a requirement specifies one or more contracts for one or more interface entities, such as operations, attributes, or operation parameters. If a requirement is stated without an associated entity, the requirement is a *default requirement* that applies by default to all entities within the interface in question. Our intention is that the default contract is the strongest contract that applies to *all* entities within an interface. We can then explicitly specify a stronger contract for individual entities by using the

<i>profile</i>	::=	<b>profile</b> { <i>req</i> <sub>1</sub> ; ...; <i>req</i> <sub><i>n</i></sub> };
<i>req</i>	::=	<b>require</b> <i>contractList</i>
		<b>from</b> <i>entityList</i> <b>require</b> <i>contractList</i>
<i>contractList</i>	::=	<i>conExp</i> <sub>1</sub> , ..., <i>conExp</i> <sub><i>n</i></sub>
<i>entityList</i>	::=	<i>entity</i> <sub>1</sub> , ..., <i>entity</i> <sub><i>n</i></sub>
<i>entity</i>	::=	<i>opName</i>
		<i>attrName</i>
		<i>opName.parName</i>
		<b>result of</b> <i>opName</i>
<i>opName</i>	::=	<i>identifier</i>
<i>attrName</i>	::=	<i>identifier</i>
<i>parName</i>	::=	<i>identifier</i>
<i>conExp</i>	::=	<u>defined in Figure 10</u>

Figure 13. Abstract syntax for profiles

<i>declaration</i>	::=	<i>conTypeDecl</i>
		<i>conDecl</i>
		<i>profileDecl</i>
<i>profileDecl</i>	::=	<i>x</i> <sub><i>p</i></sub> <b>for</b> <i>intName</i> = <i>profileExp</i>
<i>profileExp</i>	::=	<i>profile</i>
		<i>x</i> <sub><i>p</i></sub> <b>refined by</b> { <i>req</i> <sub>1</sub> ; ...; <i>req</i> <sub><i>n</i></sub> };
<i>intName</i>	::=	<i>identifier</i>
<i>conTypeDecl</i>	::=	<u>defined in Figure 10</u>
<i>conDecl</i>	::=	<u>defined in Figure 10</u>
<i>profile</i>	::=	<u>defined in Figure 13</u>
<i>req</i>	::=	<u>defined in Figure 13</u>

Figure 14. Abstract syntax for definition of profiles

refinement mechanism.

Contracts for individual entities are defined as follows: “**from** *e* **require** *C*.” Here *e* is an entity and *C* is a contract. We use *C* as a delta that refines the default contract of the enclosing profile. Using individual entity contracts as deltas for refinement means that we do not have to repeat the default QoS constraints as part of each individual contract.

Although a profile refers to specific operations and arguments within an interface, the final association between the profile and the interface is established in a profile definition. Such definitions are described in section 4.6.

For each contract type, such as reliability, that a profile involves, we may specify zero or one default contract. In addition, at most one contract of a given type can be explicitly associated with an interface entity.

If, for a given contract type *T*, there is no default contract and there is no explicit specification for a particular interface entity, the semantics is that no QoS properties within the category of *T* are associated with that entity.

```

interface NameServer {
  void init();
  void register(in string name, in object ref);
  object lookup(in string name);
}

nameServerProfile for NameServer = profile {
  require nameServerReliability;
  from lookup require Reliability contract {
    rebindPolicy == noRebind;
  };
}

```

**Figure 15.** The interface of a name server

#### 4.6. Definition of Profiles

A profile definition associates a profile with an interface and gives the profile a name. A general requirement is that the interface entities referred to by the profile must exist in the related interface. The syntax for profile definition is given in Figure 14. The definition “*id for intName = prof*” gives the name *id* to the profile which is the result of evaluating the profile expression *prof* with respect to the interface *intName*. The profile name can be used to associate this particular profile with implementations of the *intName* interface or with references to objects of type *intName*.

A profile expression (*profileExp*) can be a profile, or an identifier with a “{...}” clause. If the expression is a profile value, the definition binds a name to this value. If a profile expression contains an identifier and a “{...}” clause, the identifier must be the name of a profile, and the “{...}” clause then refines this profile. The definition gives a name to this refined profile.

If we have a profile expression “*A refined by {...}*,” then the delta must either add to the specifications in *A* or make the specifications in *A* stronger. The delta can add specifications by defining individual contracts for entities that do not have individual contracts in *A*. Moreover, the delta can specify a default contract if no default contract is specified in *A*. The delta can strengthen *A*’s specifications by giving individual contracts for entities that also have an individual contract in *A*. The individual contract in the delta are then used as a contract delta to refine the individual contract in *A*. Similarly, the delta can specify a contract delta that refines the default contract in *A*. We give a more detailed and formal description of profile refinement in [9].

To exemplify the notion of profile definition, consider the interface of a name server in Figure 15. The profile called `nameServerProfile` is a profile for the `NameServer` interface; it associates various contracts with the operations defined with the `NameServer` interface. The `nameServerProfile` associates the `nameServerReliability` contract (introduced in Figure 12) as the default contract, and it associates a refinement of the `nameServerReliability` contract with the `lookup` operation.

Notice that the contract for the `lookup` operation must refine the default contract (in this case, the default contract is `nameServerReliability`). Since the contract for



<i>binding</i>	::=	<i>clientBinding</i>   <i>serviceBinding</i>
<i>clientBinding</i>	::=	<i>refDecl</i> <b>with</b> <i>profileExp</i>
<i>serviceBinding</i>	::=	<i>serviceDecl</i> <b>with</b> <i>profileExp</i>
<i>refDecl</i>	::=	<i>identifier</i> : <i>intName</i>
<i>serviceDecl</i>	::=	<i>identifier</i> <b>implements</b> <i>intName</i>

Figure 16. Abstract syntax for bindings

```

//Client side binding
myNameServer: NameServer with nameServerProfile;

//Implementation binding
myNameServerImp implements NameServer
with nameServerProfile;

```

Figure 17. Example bindings

operations must always refine the default contract, it is implicitly understood that the contract expression in an operation contract is in fact a refinement.

#### 4.7. Bindings

There are many ways in which QoS profiles can be bound to specific services. They can be negotiated and associated with deals between clients and server, or they can be associated statically at design or deployment time. For the purpose of this paper we will provide an example binding mechanism that allows clients to statically bind profiles to references. In addition, we allow a server to state the profile of its implementation. These bindings could be used to ensure compatible characteristics for clients and servers as well as runtime monitoring. An abstract syntax for our notion of binding is illustrated in Figure 16.

Figure 17 illustrates our notion of binding. In the first example the client declares a reference called `myNameServer` as a reference to a name server. The client's QoS requirements are expressed by means of the profile called `nameServerProfile`. In the second example, the implementation called `myNameServerImp` is declared to implement the service specification that consists of the interface called `NameServer` and the profile called `nameServerProfile`.

The binding mechanism need not be a part of QML but has been included here for clarity. Bindings are more closely related to interface specification, design and implementation languages. As an example we will propose a binding mechanism for UML in section 5.

#### 4.8. Conformance

We define a conformance relation on profiles, contracts, and constraints. A stronger specification conforms to a weaker specification. We need conformance at runtime so that client-server connections do not have to be based on exact match of QoS requirements with QoS properties. Instead of exact match, we want to allow a service

to provide more than what is required by a client. Thus, we want service specifications to conform to client specifications rather than match them exactly.

Profile conformance is defined in terms of contract conformance. Essentially, a profile  $P$  conforms to another profile  $Q$  if the contracts in  $P$  associated with an entity  $e$  conform to the contracts associated with  $e$  in the profile  $Q$ .

Contract conformance is in turn defined in terms of conformance for constraints. Constraint conformance defines when one constraint in a contract can be considered stronger, or as strong as, another constraint for the same dimension in another contract of the same contract type.

To determine constraint conformance for set dimensions, we need to determine whether one subset conforms to another subset. Conformance between two subsets depends on their ordering. In some cases, a subset represents a stronger commitment than its supersets. As an example, let us consider the failure-masking dimension. If a value of a failure-masking dimension defines the failures exposed by a server, a subset is a stronger commitment than its supersets (the fewer failure types exposed, the better). If, on the other hand, we consider a payment protocol dimension for which sets represent payment protocols supported by a server, a superset is obviously a stronger commitment than any of its subsets (the more protocols supported, the better). Thus, to be able to compare contracts of the same type the dimension declarations need to define whether subsets or supersets are stronger.

A similar discussion applies to the numeric domain. Sometimes, larger numeric values are considered conceptually stronger than smaller. As an example, think of throughput. For dimensions such as latency, smaller numbers represent stronger commitments than larger numbers.

In general, we need to specify whether smaller domain elements are stronger than or weaker than larger domain elements. The **decreasing** declaration implies that smaller elements are stronger than larger elements. The **increasing** declaration means that larger elements are stronger than smaller elements. If a dimension is declared as decreasing, we map “stronger than” to “less than” ( $<$ ). Thus, a value is stronger than another value, if it is smaller. An increasing dimension maps “stronger than” to “greater than” ( $>$ ). The semantics will be that larger values are, considered stronger.

We want conformance to correspond to constraint satisfaction. For example, we want the constraint  $d < 10$  to conform to the constraint  $d < 20$ . But  $d < 10$  only conforms to  $d < 20$  if the domain is decreasing (smaller values are stronger). To achieve the property that conformance corresponds to constraint satisfaction, we allow only the operators  $\{=, <=, <\}$  for decreasing domains, and we allow only the operators  $\{=, >=, >\}$  for increasing domains. Thus, if we have an increasing domain, the constraint  $d < 20$  would be illegal.

If a profile  $Q$  is a refinement of another profile  $P$ ,  $Q$  will also conform to  $P$ . Refinement is a static operation that gives a convenient way to write QoS specifications in an incremental manner. Conformance is a dynamic operation that, at runtime, can determine whether one specification is stronger than another specification. For more details on conformance we refer to [9].

## 5. An Extension of the Unified Modeling Language

In order to make QoS considerations an integral part of the design process, design notations must provide the appropriate language concepts. We have already presented

a textual syntax to define QoS properties. Here, we extend UML [5] to support the definition of QoS properties. Later, we will use CORBA IDL [17] and our extension of UML [5] to describe an example design that includes QoS specifications.

In UML, *classes* are represented by rectangles. In addition, UML has a *type* concept that is used to describe abstractions without providing an implementation. Types are drawn as classes with a type stereotype annotation added to it. In UML, classes may implement types. The UML *interface* concept is a specialized usage of types. Interfaces can be drawn as small circles that can be connected to class symbols. A class can use or provide a service specified by an interface. The example below shows a client using (dotted arrow) a service specified by an interface called *I*. We also show a class `Implementation` implementing the *I* interface but in this example the interface circle has been expanded to a class symbol with the *type* annotation.

Our extension to UML allows QoS profiles to be associated with *uses* and *implements* relationships between classes and interfaces. A reference to a profile is drawn as a rectangle with a dotted border within which the profile name is written. This profile box is then associated with a uses or implements relationship.

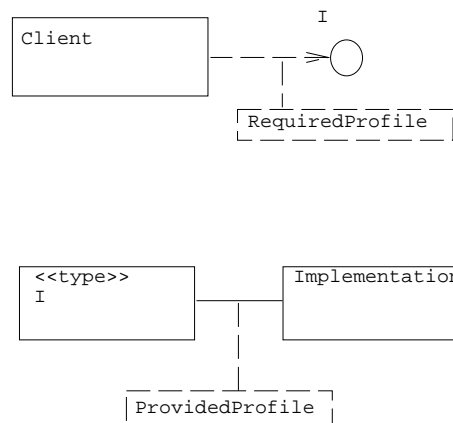


Figure 18. UML extensions

In example 18, the client requires a server that implements the interface *I* and satisfies the QoS requirements stated in the associated `RequiredProfile`. The `Implementation` on the other hand promises to implement interface *I* with the QoS properties defined by the `ProvidedProfile` profile. The profiles are defined textually using our QoS specification language.

Our UML extension allows object-oriented design to be annotated with profile names that refer to separately defined QoS profiles. Notice that our UML extension associates profiles with specific implementations and usages of interfaces. This allows different clients of the same interface to require different QoS properties, and it allows different implementations of the same interface to provide different QoS properties.

## 6. Example

To illustrate QML and demonstrate its utility, we use it to specify the QoS properties of an example system. The example shows how QML can help designers decompose

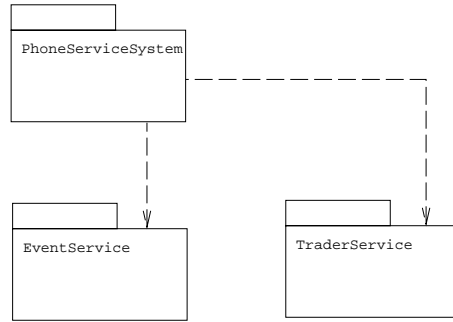


Figure 19. High-level architecture

application level QoS requirements into QoS properties for application components. The example also demonstrates that different QoS trade-offs can give rise to different designs.

This example is a simplified version of a system for executing telephony services, such as telephone banking, ordering, etc. The purpose of having such an execution system is to allow rapid development and installation of new telephony services. The system must be scalable in order to be useful both in small businesses and for servicing several hundred simultaneous calls. More importantly—especially from the perspective of this paper—the system needs to provide services with sufficient availability.

Executing a service typically involves playing messages for the caller, reacting to key strokes, recording responses, retrieving and updating databases, etc. It should be possible to dynamically install new telephone services and upgrade them at runtime without shutting down the system. The system answers incoming telephone calls and selects a service based on the phone number that was called. The executed service may, for example, play messages for the caller and react to events from the caller or events from resources allocated to handle the call.

Telephone users generally expect plain old telephony to be reliable, and they commonly have the same expectations for telephony services. A telephony service that is unavailable will have a severe impact on customer satisfaction, in addition, the service company will lose business. Consequently, the system needs to be highly available.

Following the categorization by Gray et al. [11], we want the telephony service to be a *highly-available* system which means it should have a total maximum down-time of 5 minutes per year. The availability measure will then be 0.99999. We assume the system is built on a general purpose computer platform with specialized computer telephony hardware. The system is built using a CORBA [17] Object Request Broker (ORB) to achieve scalability and reliability through distribution.

### 6.1. System Architecture

We call the service execution system module `PhoneServiceSystem`. As illustrated by Figure 19, it uses an `EventSystem` module and a `TraderService` module.

Opening up the `PhoneServiceSystem` module in Figure 20, we see its main classes and interfaces. Classes are drawn as rectangles and interfaces as circles.

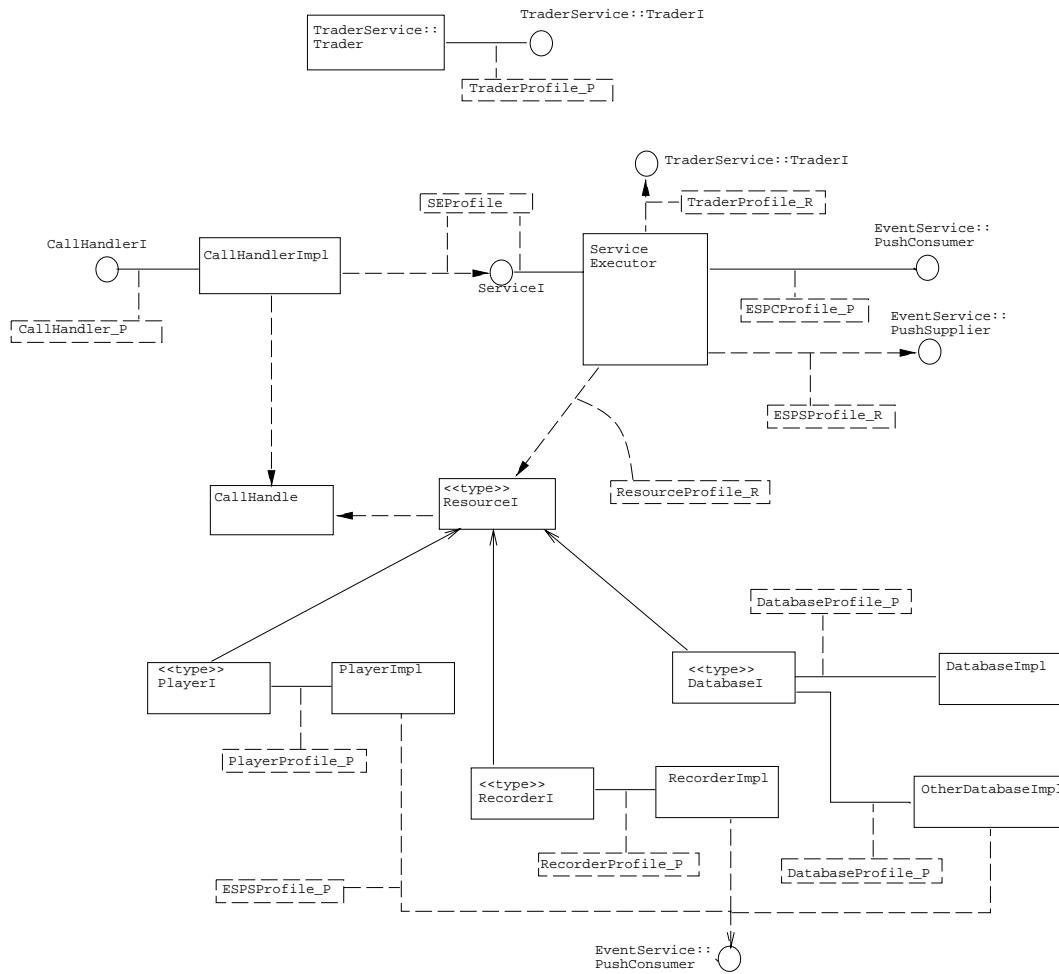


Figure 20. Class diagram for PhoneServiceSystem

Classes implement and use interfaces. As an example, the diagram shows that **ServiceExecutor** implements **ServiceI** and uses **TraderI**. In the diagram we have included references to QML profiles—such as **PlayerProfile\_P**—of which a subset will be described in section 6.2. To ease the reading of the diagram we have named *required* and *provided* profiles so that they end with the letters *R* and *P* respectively. We have omitted to draw some interrelationships for the purpose of keeping the diagram simple.

**CallHandlerI**, **ServiceI**, and **ResourceI** are three important interfaces of the system. The model also shows that the system uses interfaces provided by the **EventService** and **TraderService**.

When a call is made, the **CallHandlerImpl** receives the incoming call through the **CallHandlerI** interface and invokes the **ServiceExecutor** through the **ServiceI** interface. **CallHandlerImpl** receives the telephone number as an argument and maps that to a service identity. When **CallHandlerImpl** calls the **ServiceExecutor** it supplies the service identifier as an argument and a **CallHandle**. The **CallHandle**

```

interface ServiceI {
    void execute(in ServiceId si, in CallHandle ch) raises (InvalidSI);
    boolean probe() raises (ProbeFailed);
};

```

**Figure 21.** The ServiceI interface

contains information about the call—such as the speech channel—that is needed during the execution of the service. A new instance of `CallHandle` is created and initialized by the `CallHandler` when an incoming call is received. The information in the `CallHandle` remains unchanged for the remainder of the call.

In order to execute a service, the `ServiceExecutor` retrieves the service description associated with the received service identifier. It also needs to allocate resources such as databases, players, recorders, etc. To obtain resources, the `ServiceExecutor` calls the `Trader`. Each resource offer its services when it is initially started by contacting the trader and registering its offer. To reduce complexity of the diagram we omit showing that resources use the trader.

`ServiceExecutor` uses the `PushSupplier` and implements the `PushConsumer` interface in the `EventService` module. Resources connect to the event service by using the `PushConsumer` interfaces. The communication between the service executor and its resources is asynchronous. When the service executor needs a resource to perform an operation, it invokes the resource which returns immediately. The service executor will then continue executing the service or stop to wait for events. When the resource has finished its operation, it notifies the service executor by sending an event through the event service. This communication model allows the service executor to listen for events from many sources at the same time, which is essential if, for example, the service executor simultaneously initiates the playing of menu alternatives and waits for responses from the caller.

Figure 20 also includes references to QoS profiles. In new designs, clients and services are usually designed to match each others needs therefore the same profile often specifies both what clients expect and what services provide. When clients and services refer to the same profiles, it becomes trivial to ensure that the requirements by a client are satisfied by the service. To point out an example, `CallHandlerImpl` requires that the `ServiceI` interface is implemented with the QoS properties defined by `SEProfile_P` and at the same time `ServiceExecutor` provides `ServiceI` according to the same QoS profile.

In other cases, such as the `Trader`, are expected to preexist and therefore have previously specified QoS properties. In those situations we have one contract specifying the required properties and another contract specifying what is provided. Consequently we need to make sure the provided characteristics satisfy the required; this is referred to as *conformance* and is discussed in section 4.8.

We will now present simplified versions of three main interfaces in the design. The `ServiceI` interface provides an operation, called `execute`, to start the execution of a service. The service identifier is obtained from a table that maps phone numbers to services. The `CallHandle` argument contain channel identifiers and other data necessary to execute the service.

The `Trader` allows resources to offer and withdraw their services. Service executors can invoke the `find` or `findAll` operations on the `Trader` to locate the

```

interface TraderI {
  OfferId offer(in OfferRec or, in Object obj) raises (invalidOffer);
  Match find(in Criteria cr) raises (noMatch);
  MatchSeq findAll(in Criteria cr) raises (noMatch);
  void withdraw(in OfferId o) raises (noMatch);
};

```

Figure 22. The TraderI interface

```

interface PlayerI : ResourceI {
  void play(in CallHandle ch, in MsgSeq ms) raises (InvalidMsg);
  void stop(in CallHandle ch);
};

```

Figure 23. The PlayerI interface

resources they need. Using a trader allows us to decouple `ServiceExecutors` and resources. This decoupling make it possible to smoothly introduce new resources and remove malfunctioning or deprecated resources. Observe that this is a much simplified trader for the purpose of this paper.

Finally, we have the `PlayerI` that represents a simple player resource. Players allow us to play a sequence of messages on the connection associated with the supplied `CallHandle`. The idea is that a complete message can be built up by a sequence of smaller phrases. The interface allows the service executor to interrupt the playing of messages by calling `stop`.

## 6.2. Reliability

We have already shown in Figure 20 how profiles are associated with *uses* and *implements* relationships between interfaces and classes. We will now in more depth discuss what the QoS profiles and contracts should be for this particular design. For the contracts we will use the dimensions proposed in section 3. We will not present any development process with which you identify important profiles and their content.

To meet end-to-end reliability requirements, the underlying communications infrastructure, as well as the execution system, must meet reliability expectations. We assume that the communications infrastructure is reliable, and focus on the reliability of the service execution system.

From a telephone user's perspective, the interface `CallHandlerI` represents the peer on the other side of the line. Thus, to provide high-availability to telephone users, the `CallHandlerI` service must be highly-available.

To provide a highly-available telephone service, we require that the `CallhandlerImpl` has very short recovery time and long time between failures. Due to the expected shopping behavior of telephone service users we must require the repair time (MTTR) to not significantly exceed 2 minutes and that the variance is small.

The `CallHandler` does not provide any sophisticated failure masking, but it has a special kind of object reference that does not require rebinding after a failure. We are prepared to accept on average 2 failures per year. If the service fails, any executing and pending requests are discontinued and removed. This means we have a **at most**

```

CallServerReliability = Reliability contract {
  MTTR {
    percentile 100 <= 2;
    variance <= 0.3
  };
  TTF {
    percentile 100 > 0.05 days;
    percentile 80 > 100 days;
    mean >= 140 days;
  };
  availability >= 0.99999;
  contAvailability >= 0.99999;
  failureMasking == { omission };
  serverFailure == initialState;
  rebindPolicy == noRebind;
  numOffFailure <= 2 failures/year;
  operationSemantics == atMostOnce;
};

CallHandlerProfile_P for CallHandlerI = profile {
  require CallServerReliability;
}

```

Figure 24. Contract and binding for CallHandler

once operation semantics. The contract and profile of `CallHandlerI` as provided by `CallHandlerImpl` is described in Figure 24.

From Figure 20 we can see that the reliability of `CallHandlerI` directly depends on the reliability of service defined by `ServiceI`. `ServiceExecutor` can not provide any services without resources. Unless `ServiceExecutor` can handle failing traders and resources the reliability depends directly on the reliability of `TraderI` and any resources it uses. In this example we want to keep the `ServiceExecutor` as small and simple as possible, therefore we propagate high-availability requirements from `CallHandlerI` to the trader and the resources. This is certainly a major design decision which will affect the design and implementation of the other components of the system.

We expect the `ServiceExecutor` to have a short recovery time since it holds no information that we wish to recover. If it fails, the service interactions it currently executes will be discontinued. We assume that users consider it more annoying if a session is interrupted due to a failure than if they are unable to connect to the service. We therefore require the `ServiceExecutor` to be reliable in the sense that it should function adequately over the duration of a typical service call. Calls are estimated to last 3 minutes on average with 80% of the calls less than 5 minutes. With this in mind, we will require that the service executor provides high continuous availability with a time period of 5 minutes.

Since the recovery time is short, we can allow more frequent failures without compromising the availability requirements.

The `ServiceExecutor` recovers to a well defined `initial state` and will forget about all executions that where going on at the time of the failure. The contract



```

ServiceExecutorReliability = Reliability contract
{
  MTR < 20 sec;
  TTF {
    percentile 100 > 0.05 days;
    percentile 80 > 20 days;
    mean > 24 days;
  }
  availability >= 0.99999;
  contAvailability > 0.999999 ;
  failureMasking == { omission };
  serverFailure == initialState;
  rebindPolicy == rebind;
  numOffFailure <= 10 failures/year;
  operationSemantics == atMostOnce;
};

SEProfile for ServiceI = profile {
  require ServiceExecutorReliability;
  require Reliability contract { dataPolicy == invalid; };
};

```

Figure 25. Contract and binding for service

states that rebinding is necessary, which means that when the service executor is restarted, the `CallHandler` receives a notification that it can obtain a reference to the `ServiceExecutor` by rebinding. Pending requests are executed at most once in case of a failure; most likely they are not executed at all which is considered acceptable for this system. The contract and profile used for `ServiceI` are described in Figure 25.

Although the `ServiceExecutor` itself can recover rapidly, it still depends on the `Trader` and the resources.

We expect the `Trader` to have a relatively short recovery time, which relaxes the mean time to failure requirements slightly. We insist that all types of telephony services can be executed when the system is up, which means that all resources must be available and consequently satisfy the high-availability requirements.

The reliability contract for the `Trader` (Figure 27) is based on a general contract (`HAServiceReliability`) for highly-available services. The contract is abstract in the sense that it only states the availability requirements and leaves several of the other dimensions unspecified. The `Trader` profile refines it by stating that the recovery time should be short.

In addition, we state that offer identifiers and object references returned by the trader are valid even after a failure. This means that an offer identifier returned before a failure can be used to withdraw an offer after the `Trader` has recovered. Also, any references returned by the `Trader` are valid during the `Trader`'s down period as well as after it has recovered, assuming, of course, that the services referred to by the references have not failed.

The start-up time for a service execution is very important; the time between a call is answered and the service starts executing must be short and definitely not more than one second. A start-up time that exceeds one second can make users believe there

is a problem with the connection and therefore hang-up the phone, the consequence being both an unsatisfied customer and a lost business opportunity.

Having analyzed and estimated the execution times in the start-up execution path, we require that the `find` and `findAll` operations on the `Trader` respond quickly. We do not anticipate the throughput to constitute a bottleneck in this case.

We can relax the performance requirements for the `offer` and `withdraw` operations on the `Trader`. The reason being that these operations are not time critical from the service execution point of view. We specify the performance in Figure 27 as part of the `TraderProfile_P` profile.

The performance profile makes it clear that the implementation of `TraderI` should give invocations of `find` and `findAll` higher priority than invocations of `offer` and `withdraw`.

A resource service represents a pool of hardware and software resources that are expected to be highly-available. If a resource service is down, it is likely that there are major hardware or software problems that will take a long time to repair. Since failing resource services are expected to have long recovery times, they need to have, in principle, infinite MTTF to satisfy high availability requirements. This does not mean that individual resource cannot fail, but it does mean that there must be sufficient redundancy to mask failures.

In Figure 26 we define a general contract, called `ResourceReliability`, for `ResourceI`. The contract captures that resources need to be highly available. Each specific resource type—such as `PlayerReliability`—will then refine this general contract to specify its individual QoS properties.

### 6.3. Discussion

The specification of reliability and performance contracts, and the analysis of inter-component QoS dependencies, have given us many insights and important guidance. As an example, it has helped us realize that the `Trader` needs to support fast fail-over and use a reliable storage. We also found that the reliability of resources is essential, and that, in this example system, resource services should be responsible for their own reliability. The explicit specification also allows us to assign well-defined values to various dimension which make design goals and requirements more clear.

QML allows detailed descriptions of the QoS associated with operations, attributes, and operation parameters of interfaces. This level of detail is essential to clearly specify and divide the responsibilities among client and service implementations. The refinement mechanism is also essential. Refinement allows us to form hierarchies of contracts and profiles, which allows us to capture QoS requirements at various levels of abstraction.

Due to the limited space of this paper, we have not been able to include a full analysis or specification of the example system. In a real design, we also need to study what happens when various components fail, estimate the frequency of failures due to programming errors, etc. We also need to ensure that the QoS contracts provided by components actually allows the clients to satisfy requirements imposed on them. There are various modeling techniques available that are applicable to selected types of systems; see Reibman et al. [19] for an overview.

In our case, high availability requirements for `CallHandler` have resulted in strong demands on other services in the application. Another design alternative would be to demand that components such as the `ServiceExecutor` can handle failing

```

ResourceReliability = Reliability contract {
  availability >= 0.99999;
  failureMasking == { failure };
  serverFailure == initialState;
  rebindPolicy == rebind;
};

PlayerReliability =
  ResourceReliability refined by {
    MTR = 7200 sec;
    TTF {
      percentile 100 > 2000 days;
      percentile 80 > 6000 days;
      mean >= 7000 days;
    };

    availability >= 0.99999;
    contAvailability >= 0.999999;
    failureMasking == failure;
    serverFailure == initialState;
    rebindPolicy == rebind;
    numOfFailure <= 0.1 failures/year;
    operationSemantics == least_once;
    dataPolicy == no_guarantees;
  };

PlayerProfile_P for PlayerI = profile {
  require PlayerReliability;
};

```

Figure 26. Contract and binding for resources

resources and switch to other resources when needed. This would require more from the `ServiceExecutor`, but allow resource services to be less reliable.

Despite the limitations of our example, we believe that it demonstrates three important points: QoS should be considered during the design of distributed systems; QoS requires appropriate language support; QML is useful as a QoS specification language.

Firstly, we want to stress that considering QoS during design is both useful and necessary. It will directly impact the design and make developers aware of non-functional requirements.

Secondly, QoS cannot be effectively considered without appropriate language support. We need a language that helps designer capture QoS requirements and associate these with interfaces at a detailed level. We also need to make QoS requirements and offers first class citizens from a design language point of view.

Finally, we believe the example shows that QML is suitable to support designers in involving QoS considerations in the design phase.

```

HServiceReliability = Reliability contract {
  availability >= 0.99999;
  failureMasking == { omission };
  serverFailure == initialState;
  rebindPolicy == rebind;
  numOfFailure <= 10 failures/year;
  operationSemantics == once;
};

TraderProfile_P for TraderI = profile {
  require HServiceReliability refined by {
    MTRR {
      percentile 100 < 60 ;
      variance <= 0.1;
    }
  };

  from offer.OfferId, result of find, findAll require Reliability
  contract { dataPolicy == valid; };

  from find, findAll require Performance
  contract { latency { percentile 90 < 50 }; };

  from offer, withdraw require Performance
  contract { latency { percentile 80 < 2000 }; };
};

```

Figure 27. Contract and binding for the Trader

## 7. QoS Specifications at Runtime

### 7.1. QoS-aware Systems

So far in this paper, we have used QoS specifications to create abstraction boundaries between objects. We used QML to extend the traditional notion of interface to cover non-functional properties. However, to build systems that deliver predictable QoS, it is often necessary to use QoS specifications as part of object implementations as well. We introduce the notion of *QoS-aware* objects, which means that objects know which level of QoS they require and provide, and that they are able to communicate this knowledge to other objects. As we elaborate in the following, QoS awareness is necessary if we want to deploy objects in open systems or if we want to monitor compliance of QoS contracts at runtime.

The configuration of an *open* system may change over time. New services may be dynamically added or removed, and the connection between clients and services may change over time. Another characteristic of an open system is that no single entity controls system evolution. The Internet is an example of an open system. In an open system, we cannot a priori set up QoS contracts between clients and services. We do not know which services will be available over time, nor do we know which level

of QoS these services will provide. In open systems, we want to make clients and services QoS aware so that client-server connections can be established dynamically based on client QoS requirements and service QoS properties. In contrast, we say a system is *closed* when the structure and interconnections are defined statically during configuration or deployment. It is then the task of the configuration to match objects so that QoS requirements for the system as a whole are met.

Consider the currency trading system described in Section 1. In an open system the trading station would be willing to connect to any rate service as long as it provides adequate QoS. In order to select an object the trading station could communicate the QoS it requires as well as its own QoS characteristics to an intermediate broker. The broker will compare—by using the QML conformance relation—the clients specifications with those of objects offering the `RateServiceI` interface. When matching offers are found the broker will connect the client and the server objects.

QoS-aware objects may use statically constructed QoS specifications or create specifications dynamically based on environmental and end-user requirements. A QoS-aware object may provide multiple modes of operation each offering different QoS depending on the QoS it receives from other objects and its run-time environment. To support QoS awareness, we need more than a specification language, we need to support QoS specifications as first-class entities that can be dynamically created, communicated, and manipulated.

Another example of the benefits of QoS-awareness is illustrated by the example in section 6. In the example we describe how QML helps us break down the QoS requirements for the system as a whole to QoS specifications for individual objects. The focus of the example is on static specification where we implicitly assumed a closed system. Let us instead assume that the system is open. This could for example manifest itself in that resources can be added dynamically. To satisfy QoS requirements the service executor will need to request resources with specific QoS characteristics. The QoS of resources must be explicitly specified and we need to be able to compare specifications dynamically. This again, will require that we have the means for creating, manipulating, comparing, and communicating QoS specifications as runtime entities.

It is very hard—and in most cases impossible—to verify statically that a component in will fact provide specified QoS. An alternative to verification is to monitor requests between objects and determine whether the specified QoS is satisfied. Monitoring itself imposes its own set of issues that we consider outside the scope of this paper. An interesting aspect is, however, that monitoring requires runtime representations of QoS specifications.

To support construction of open systems with predictable QoS, and to support compliance monitoring of QoS contracts, we create a QoS fabric where QoS specifications are first-class entities at runtime. In the next section we briefly describe implementation of this fabric.

### 7.2. QRR: A QML-Based QoS Fabric

Besides being efficient and scalable, we have the following requirements for our QRR QoS runtime fabric:

- (i) QRR should support the same fundamental concepts as QML. We want to use the same QoS specification concepts during design and implementation. Using the same concepts also implies that the precise, formal definition of QML carries over

to QRR. A precise definition will in turn significantly improve interoperability of different QRR/QML components.

- (ii) It should be possible to easily create new QRR specifications at runtime by calling generic library functions. Since the QoS requirements of an application may not be known at compile time, we do not want to insist that all QRR specifications arise from compiled QML specifications.
- (iii) Although QML is purely declarative and does not support specifications that change over time, we need to support runtime manipulation of QRR specifications. For some QoS categories, such as performance, the level of QoS that a service can provide depends on its dynamic runtime environment and the availability of resources in that environment. To enable specifications that reflect the dynamic availability of resources, we need to support runtime manipulation of QRR specifications.

We are implementing QRR to satisfy these requirements. Currently, we have implemented a prototype QML compiler and a prototype QRR library. We have successfully compiled QML specifications into QRR, instantiated those specifications in a CORBA environment, communicated the specifications between distributed components, and compared them using a conformance checking function that is part of the QRR library. The following section describes the basic implementation principles in more detail.

### 7.3. Implementation Architecture

This section presents the main principles behind the implementation of QRR. The reader may wish to consult [10] for more details on the implementation and for examples of its use.

The QRR implementation consists of a C++ library that allows applications to create QRR specifications and to check conformance of these specifications. Some of the data types used by the library are generated from CORBA IDL type definitions to facilitate the communication of QRR specifications between distributed CORBA objects. The QRR compiler emits a mix of IDL and C++ code to represent a particular QML specification. The emitted IDL code consists of types that represent the QML specification. The C++ code contains functions to create QRR instances of the QML specification.

In the following, we briefly describe how we represent the various QML constructs in a mixture of CORBA IDL and C++. Profiles are defined as instances of the `profile` struct shown in Figure 28. They contain the profile name, interface name, a sequence of default contracts (`dcontracts`), and a sequence (`profs`) of structs, each associating an entity with a set of contracts. The `profs` sequence represents the individual contracts of the profile. In QRR, all profiles are instances of the `profile` struct. For a particular profile specified in QML, the QML compiler emits a C++ function that constructs an instance of the `profile` struct.

QoS constraints are represented as instances of the struct `constraint` in Figure 29. A `constraint` struct has a sequence of `aspect` structs as well as a tag indicating whether it is a simple constraint—such as “*delay* < 10”—or a set of aspects representing statistical characterizations. We define a separate struct type for each aspect kind, however the figure only shows the struct used to represent mean aspects.

```

struct profile {
    string pname;
    string iname;
    contractSeq dcontracts;
    entityProfileSeq profs;
};

```

Figure 28. IDL for profile

```

enum aspectKind {
    ak_freq, ak_perc,
    ak_mean, ak_var,
    ak_simple
};

struct mean {
    operators op;
    value num;
};

struct aspect {
    aspectKind ak;
    any asp;
};

typedef sequence <aspect> aspects;

struct constraint {
    constrKind ck;
    aspects asps;
};

```

Figure 29. IDL for aspect and constraint

Because IDL does not allow polymorphism for structs, we wrap aspect instances in *any* type along with a type tag.

We provide two alternative representations for contracts and contract types. In the *generic* representation, all contracts are instances of the same type, and this type is then part of the QRR library. In the *static* representation, only contracts of the same QML contract type are instances of the same QRR type. In addition, the QRR types used for the static representation are emitted by the QML compiler.

The static representation requires that the emitted QRR types are linked into the application that instantiates them. On the other hand, using distinct QRR types for distinct contract types facilitates a more efficient implementation of conformance checking and other QRR functions.

With the generic representation, applications can dynamically create and communicate contracts whose types are not known at compile time. Although we describe them as separate representations, our goal is to allow their simultaneous use to achieve maximum flexibility and performance. We only describe the static

```

type Reliability = contract {
  numberOfFailures: decreasing numeric;
  TTR: decreasing numeric;
  availability: increasing numeric;
};

struct Reliability_i {
  tid ct;
  numberOfFailures constraint;
  TTR constraint;
  availability constraint;
};

```

Figure 30. IDL for statically generated contracts

representation in the following.

With the static representation, each QML contract type will result in the generation of two things: a C++ description of the contract type and an IDL struct used to instantiate contracts of that type. The emitted contract type description inherits from, and adds to, a set of contract type base classes implemented in the QRR library. The emitted type description contains information about the various dimensions declared as part of the contract type. For each ordered, user-defined domain, the emitted code contains a function that computes the domain ordering. For each contract type, the emitted code also contains a function for checking conformance between contracts of that type.

In addition to the emitted C++ code, the compiler also emits an IDL struct definition for each contract type. The name of this struct is the contract type name with `_i` appended to it. The struct has one field for each dimension. Each field has the same name as the corresponding dimension and is of type `constraint`.

In Figure 30 we show a QML contract type called `Reliability` and the corresponding emitted IDL struct.

An instance of the `Reliability_i` struct will hold instances of constraints that in turn hold the aspects specified for each individual constraint. An instance also contains the type identifier of its contract type. Currently, we represent type identifiers as text strings. Notice that QML has a flat name space for contract types.

The programmer can manually instantiate instances of contract structs, such as `Reliability_i`. Manual instantiation is tedious because the programmer must explicitly create structs that represent the constraints of the contract struct. To automate the instantiation process, the QML compiler emits instantiation functions for each contract and profile declared in QML.

When an application needs to check conformance, it invokes the library function `conformsTo` whose signature is shown in Figure 31. This function takes two profiles, and checks conformance between their contracts. Inside profiles, contracts are stored as a pair consisting of a contract type name and an element of type *any*. For a performance contract, the *any* element will contain an instance of type `Performance_i` and the contract type name will be the string "Performance". To check conformance between performance contracts, the `conformsTo` will use the string "Performance" to lookup the C++ object which represents performance contract types at runtime.



```

int conformsTo(profile &stronger,profile &weaker);

int Performance::conformsTo(CORBA::Any *stronger,CORBA::Any *weaker);

int checkSem(profile &p);

template <class elemType>
aspect * qml_perc_asp(operators op,int pe,elemType val);

template <class elemType>
aspect * qml_freq_asp(operators op,int fr,rangeType l,elemType low,
    elemType high,rangeType r);

```

Figure 31. Some library function signatures

This object is of type `Performance` and will have a virtual function called `conformsTo` (the signature of this function is given in Figure 31 as `Performance::conformsTo`). The `Performance::conformsTo` function is emitted. It expects two *any* arguments that both contain instances of the struct `Performance_i`. Since it is emitted, the `Performance::conformsTo` function knows which objects to extract from the *any* arguments.

Figure 31 also shows the signatures of some of the functions provided by the library to dynamically create and check QRR specifications. As an example, the `checkSem` function will check the static semantics of a profile instance.

To dynamically create profiles, we need to create contracts and aspects and build the appropriate runtime structure. The construction of such runtime structures can involve many operations and be error prone. We therefore intend to provide more convenient programming abstractions in the future. The current library implementation only provides convenience functions for individual aspects. The functions `qml_perc_asp` and `qml_freq_asp` are examples of such functions for percentile and frequency aspects respectively.

#### 7.4. Small Example

To give intuition about how to use QRR specifications, we describe a simple QoS compatibility-checking mechanism that allows a client to ask a server whether its requirements are compatible with what the server can provide.

First, let's assume that a programmer is implementing a server *A* that provides an interface  $I_1$  and uses a server *B* that implements an interface  $I_2$ . Since we are concerned with applications that are developed to meet and adapt to QoS requirements, we also assume that the programmer will specify the QoS for servers that implement  $I_1$  and  $I_2$  using QML.

To support the QoS-checking mechanism the server implements the interface `QoSAware`, which we describe in Figure 32. The operation `compatible` allows the client to send the profile it requires to the server. The server responds with `true` if the client's requirements and the server's capabilities are compatible; and with `false` otherwise. If the profile is semantically invalid, the operation raises an exception.

In our small example, we can describe—in QML—the requirements of server *A* on server *B* as a profile for the interface  $I_2$ . We can also describe the QoS provided

```

interface QoSaware {
    exception invalidProfile{};
    boolean compatible(in profile p)
        raises (invalidProfile);
};

```

Figure 32. QoSaware interface

```

CORBA::Environment env;
profile * p = i2_prof();
if (I2ref->compatible(*p,env) {
    //OK to use this server
    ....
} else {
    //use another server
    ....
};

```

Figure 33. Client call

by  $A$  as a profile for interface  $I_1$ . Having defined those profiles and the contracts that they use we can emit QRR code that can be compiled and linked with the client and server respectively.

We can create the specified profiles in server  $A$  by invoking the emitted functions that have the same names as the profiles specified in QML. If we have a profile named `i2_prof` specifying  $A$ 's requirements on  $I_2$ ,  $A$  objects would use an emitted function called `i2_prof` to create an QRR instance of this profile. The C++ code in Figure 33 illustrates how a profile can be created and sent with an ordinary CORBA request.

The implementation of `compatible` simply takes the profile specified for the server and checks its conformance to the profile supplied by the client. The implementation checks the static semantics of the profile before doing performance checking. In the future we intend to include information in a profile that allows a program to determine whether a profile has already been checked for semantic validity or not. With this extra information, we can avoid redundant semantic checks. Figure 34 describes a simple implementation of a server that supports the `QoSaware` interface.

In the example, the client directly communicated its requirements to the server. However, in many situations, QoS specifications are not communicated directly between clients and servers. Instead, the specifications tend to be used by third party components, such as monitoring or negotiation mechanisms. The third party component could also be a management service or anything else that requires to know about and can influence QoS requirements and agreements. We believe QML and QRR are useful for a wide range of uses in QoS-aware distributed object systems.

## 8. Related Work

Common object-oriented analysis and design languages, such as UML [5], Objectory [12], Booch notation [4], and OMT [22], generally lack concepts and constructs for QoS specification. In some cases, they have limited support to deal

```

CORBA::Boolean B_serverImpl::compatible(profile &p,CORBA::Environment &ev)
{
    if (! checkSem(p) {
        throw QoSAware::invalidProfile();
    };

    if(conformsTo(myprof(),p)){
        cout << "Conformance... " << endl;
        return 1;
    }
    else {
        cout << "Non-conformance..." << endl;
        return 0;
    }
};

```

**Figure 34.** Server implementation

with temporal aspects or call semantics [4].

Interface definition languages, such as OMG IDL [17], specify functional properties and lack any notion of QoS. TINA ODL [25] allows the programmer to associate QoS requirements with streams and operations. A major difference between TINA ODL and our approach is that they syntactically include QoS requirements within interface definitions. Thus, in TINA ODL, one cannot associate different QoS properties with different implementations of the same functional interface. Moreover, TINA ODL does not support refinement of QoS specifications, which is an essential concept in an object-oriented setting.

Similarly, Becker and Geihs [1] extend CORBA IDL with constructs for QoS characterizations. Their approach suffers from the same problem as the TINA ODL approach: they statically bind QoS characterizations to interface definitions. They also allow QoS characteristics to be associated only with interfaces, not individual operations. In addition, they support only limited domains and do not allow enumerations or sets. Finally, they allow inheritance between QoS specifications, but it is unclear what constraints they enforce to ensure conformance. QoS specifications are exchanged as instantiations of IDL types without any particular structure.

There are a number of languages that support QoS specification within a single QoS category. The SDL language [14] has been extended to include specification of temporal aspects. The RTSynchronizer programming construct allows modular specification of real-time properties [20]. These languages are all tied to one particular QoS category. In contrast, QML is general purpose; QoS categories are user-defined types in QML, and can be used to specify QoS properties within arbitrary categories.

The specification and implementation of QoS constraints have received a great deal of attention within the domain of multimedia systems. In [21], QoS constraints are given as separate specifications in the form of entities called QoS Synchronizers. A QoS Synchronizer is a distinct entity that implements QoS constraints for a group of objects. The use of QoS Synchronizers assumes that QoS constraints can be implemented by delaying, reordering, or deleting the messages sent between objects in the group. In contrast to QML, QoS Synchronizers not only specify the QoS constraints, they also enforce them. The approach in [24] is to develop specifications

of multimedia systems based on the separation of content, view, and quality. The specifications are expressed in  $Z$ . The specifications are not executable per se, but they can be used to derive implementations. In [2], multimedia QoS constraints are described using a temporal, real-time logic, called QTL. The use of a temporal logic assumes that QoS constraints can be expressed in terms of the relative or absolute timing of events. Campbell [6] proposes pre-defined C-language structs that can be instantiated as QoS specifications for multimedia streams. The expressiveness of the specifications are limited by the C language, thus there is no support for statistical distributions. Campbell does, however, introduce separate attributes for capturing statistical guarantees. It should be noted that Campbell does not claim to address the general specification problem. In fact, he identifies the need for more expressive specification mechanisms that include statistical characterizations. In contrast to QML, the multimedia-specific approaches only address QoS within a single domain (multimedia). Moreover, these approaches tend to assume stream-based communication rather than method invocation.

Zinky et al. [26, 27] present a general framework, called QuO, to implement QoS-enabled distributed object systems. The notion of a *connection* between a client and a server is a fundamental concept in their framework. A connection is essentially a QoS-aware communication channel; the expected and measured QoS behaviors of a connection are characterized through a number of *QoS regions*. A region is a predicate over measurable connection quantities, such as latency and throughput. When a connection is established, the client and server agree upon a specific region; this region captures the expected QoS behavior of the connection. After connection establishment, the actual QoS level is continuously monitored, and if the measured QoS level is no longer within the expected region, the client is notified through an upcall. The client and server can then adapt to the current environment and re-negotiate a new expected region.

QuO does not provide anything corresponding to refinement, conformance, or fine-grained characterizations provided by QML.

Within the Object Management Group (OMG) there is an ongoing effort to specify what is required to extend CORBA [17] to support QoS-enabled applications. The current status of the OMG QoS effort is described in [18], which presents a set of questions on QoS specification and interfaces. We believe that our approach provides an effective answer to some of these questions.

## 9. Discussion

Developing a QoS specification language is only the first step towards supporting QoS considerations in general and, as this paper suggest, as an integral part of the design process. We need methods that address the process aspects of designing with QoS in mind. For example, we need methods that help the designer make QoS-based trade-offs, and methods that help the designer decompose the application-level QoS requirements into QoS properties for individual components. In addition to methods, we also need tools that can check consistency and satisfaction of QoS specifications. For example, it would be desirable, to have a tool that can check whether a running service meets its QoS specification. Although a specification language is not a complete solution, we still believe it is an important step.

Specifying QoS properties at design time is only the starting point; eventually we need to implement the design and ensure that the QoS requirements are satisfied in the

implementation. An important issue that must be addressed in the implementation, is what action to take at runtime if the QoS requirements cannot be satisfied in the current execution environment, for example, what should happen if the actual response time is higher than the stated response time requirement. In most applications, it is not acceptable for a service to stop executing because its QoS requirements cannot be satisfied. Instead, one would expect the service to adapt to its environment through graceful degradation.

For a service to adapt to its environment, it must be notified about divergence from specified requirements, and it must be able to dynamically specify relaxed requirements to the infrastructure, and to the services it depends upon, to communicate how it can gracefully degrade and thereby adapt to the current execution environment. We believe that our concepts of profile and contract can be used to specify QoS requirements at runtime as well as at design time. To facilitate runtime specification, we need profiles and contracts to be first class values in the implementation language. To achieve this, we can define a mapping from QML into the implementation language; for example, if the implementation language is C++, one could map contract types into classes and contracts into objects instantiated from those classes. The important thing to notice is that the *concepts* remain the same.

## 10. Concluding Remarks

We argued that taking QoS into account during the design of distributed object systems significantly influences design and implementation decisions. Late consideration of QoS aspects will often lead to increased development and maintenance costs as well as systems that fail to meet user expectations.

We proposed a language, called QML, that allows developers to explicitly deal with QoS as they specify interfaces. We showed how QML can be used for QoS specification in class model and interface designs of distributed object systems. QML allows QoS specifications to be separated from interfaces and associated with uses and implementations of services. QML contains a refinement mechanism allows reuse and customization of QoS contracts. The refinement mechanism also allows us to deal with the interaction between QoS specification and interface inheritance; thus we truly support object-oriented design. We also described how we can use conformance checking to determine whether one specification satisfies another. Finally, QML allows QoS specification at a fine-grained level—operation arguments and return values—that we believe is necessary in many applications and for many QoS dimensions.

Although this paper primarily focused on the usage of QML in the context of software design, we intend to use it for the implementation and management of QoS in general. As described in Section 7, we provide programming language definitions that can be used to construct QoS specifications at runtime and treat them as first-class entities. These runtime entities are used to offer and require QoS characteristics at the application programming interface level.

Our experience suggests that the concepts and language proposed in this paper will provide a sound foundation for future QoS specification languages and integration of such languages with general object-oriented specification and design languages.

## Acknowledgments

The work presented in this paper has benefited greatly from interaction with, and feedback from, our colleagues in the Software Technology Laboratory at Hewlett-Packard Laboratories. In particular, we thank Evan Kirshenbaum for his insightful and detailed feedback. We also thank Brad Askins, Pankaj Garg, Mudita Jain, Reed Letsinger, Mary Loomis, Joe Martinka, Keith Moore, Aparna Seetharaman, and Dean Thompson. We acknowledge the feedback from Derek Coleman at Hewlett-Packard, Jørgen Nørgård at Beologic, and Christian Becker at the University of Frankfurt.

## References

- [1] C. R. Becker and K. Gheis. Maqs—Management for Adaptive QoS-Enabled Services. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December, 1997.
- [2] G. Blair, L. Blair, and J. B. Stefani. A Specification Architecture for Multimedia Systems in Open Distributed Processing. *Computer Networks and ISDN Systems*, 29, 1997. Special Issue on Specification Architecture.
- [3] K. P. Birman. ISIS: A System for Fault-Tolerant Distributed Computing. Department of Computer Science, Cornell University. TR86-744, April, 1986.
- [4] G. Booch. Object-Oriented Analysis and Design. Benjamin-Cummings Corp. 1994.
- [5] G. Booch, I. Jacobson, and J. Rumbaugh. Unified Modeling Language. *Rational Software Corporation*, version 1.0, January 1997.
- [6] A. T. Campbell. *A Quality of Service Architecture*. PhD thesis, January, 1996.
- [7] D. Coleman, P. Arnold, S. Bodoff, C. Dolin, F. Hayes, and P. Jeremaes. Object-Oriented Development: The Fusion Method. Prentice-Hall. 1994.
- [8] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, Vol. 34, No. 2, February 1991.
- [9] S. Frølund and J. Koistinen. QML: A Language for Quality-of-Service Specification. Hewlett-Packard Laboratories, Technical Report HPL-98-10, February, 1998.
- [10] S. Frølund and J. Koistinen. Quality of Service Aware Distributed Object Systems. Hewlett-Packard Laboratories, Technical Report HPL-98-142, July, 1998.
- [11] J. Gray and S. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann. 1993.
- [12] I. Jacobson, M. Christerson, and Persson. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.
- [13] J. Koistinen. Dimensions for Reliability Contracts in Distributed Object Systems. Hewlett-Packard Laboratories, Technical Report HPL-97-119, October, 1997.
- [14] S. Leue. Specifying Real-Time Requirements for SDL Specifications—a Temporal Logic-Based Approach. Protocol Specification, Testing, and Verification XV. in *Proceedings of the Fifteenth IFIP WG6*. June, 1995.
- [15] B. Littlewood. Software Reliability Modelling. In *Software Engineer's Reference book*. Section 31, Butterworth-Heinemann Ltd., 1991.
- [16] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. in *Proceedings of the USENIX Conference on Object-Oriented Technologies*. June, 1995.
- [17] Object Management Group. *The Common Object Request Broker: architecture and specification*, July 1995. revision 2.0.
- [18] Object Management Group. *Quality of Service: OMG Green paper*. Draft revision 0.4a, June 12, 1997.
- [19] A. L. Reibman and M. Veeraraghavan. Reliability Modeling: An Overview for System Designers. *IEEE Computer*, April, 1991.
- [20] S. Ren and G. Agha. RTSynchronizer: Language Support for Real-Time Specifications in Distributed Systems. *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, La Jolla, California, June, 1995.
- [21] S. Ren, N. Venkatasubramanian, and G. Agha. Formalizing multimedia qos constraints using actors. In *Proceedings of the Second IFIP International Conference on Formal Methods for Open, Object-Based Distributed Systems*, 1997.
- [22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, B. Lorensen, and W. Lorensen. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

- [23] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, Vol. 2, No. 4, November, 1984.
- [24] R. Staehlin, J. Walpole, and D. Maier. Quality-of-Service Specification for Multimedia Presentations. *Multimedia Systems*, 3(5/6), November, 1995.
- [25] TINA Object Definition Language. Telecommunications Information Networking Consortium, June 1995.
- [26] J. A. Zinky, D. E. Bakken, and R. D. Schantz. Architectural Support for Quality-of-Service for CORBA objects. *Theory and Practice of Object Systems*, Vol. 3(1), 1997.
- [27] J. A. Zinky, D. E. Bakken, and R. D. Schantz. Overview of Quality of Service for Distributed Objects. in *Proceedings of the Fifth IEEE conference on Dual Use*, May, 1995.