HEWLETT PACKARD

QML: A Language for Quality of Service Specification

Svend Frølund, Jari Koistinen
Software Technology Laboratory
HPL-98-10
February, 1998

specification languages, QoS, distributed object systems

To be competitive, future software systems must provide not only the correct functionality, but also an adequate level of quality of service (QoS). By QoS, we refer to non-functional properties, such as reliability, performance, timing, and security. To provide an adequate level of QoS, software systems need to include capabilities such as QoS negotiation, monitoring, and adaptation. These capabilities all require the expected and the provided QoS levels to be explicitly specified. QoS can be specified statically at the time of implementation, design, or dynamically at deployment or runtime.

To facilitate QoS specification, we present a general Quality of service Modeling Language (QML) for defining multi-category QoS specifications for components in distributed object systems. QML is designed to support QoS in general, encompassing QoS categories such as reliability, performance, security, and timing.

We use QML to *describe* the QoS properties of software components—QML specifications cannot be executed to implement the specified QoS. In this sense, QML is similar to interface definition languages that describe the functional properties of software components, but in contrast to interface definition languages, QML describes the non-functional properties of software components.

QoS specification in QML facilitate the static decomposition of a software system into components with precisely specified QoS boundaries. They also facilitate dynamic QoS functions, such as negotiations, monitoring, and adaptation. QML is designed for a good fit with object-oriented distributed architectures and concepts such as interfaces and inheritance. It also allows specification at a fine-grained level for operations, operation arguments, and attributes. QML enables user-defined QoS categories, and allows specifications within those categories to be associated with component interface definitions. In addition, checks can be made dynamically to determine whether one QML specification satisfies another. This mechanism allows us to dynamically match QoS requirements and offers.

Categories and Subject Descriptors: D.2.1 **[Software Engineering]**: Requirements/Specification—languages; D.3.2 **[Programming Languages]**: Language Classifications—design languages; F.3.1 **[Theory of Computation]**: Specifying and Verifying and Reasoning about Programs—specification techniques

Contents

1. INTRODUCTION

A computer system must deliver acceptable *quality of service* (QoS) to its users or its environment. By QoS, we refer to non-functional properties such as performance, reliability, availability, timing, and security. For some applications acceptable QoS is best effort; while other types of applications require guaranteed levels of QoS to function properly. In real-time systems, for example, timing is essential for correctness. In banking systems, security is necessary and must not be compromised. Although QoS has been a concern in particular domains for quite a while, there is a clear trend that users in general will increasingly require adequate and flexible QoS in addition to proper functionality. The move towards widely distributed systems makes providing desired QoS even more critical.

To design, implement, and manage distributed systems to deliver their intended QoS, we need a way to precisely specify QoS properties.

In distributed systems, clients often rely on services to provide a certain level of QoS. For example, a client may rely on a service response time of less than 50 milliseconds to provide its own response time of less than 100 milliseconds. If these QoS requirements are not explicitly specified, it is hard for software systems to evolve because it is hard to determine the QoS dependencies between the various components. Once components have precisely defined QoS properties, we can safely replace old components with new components as long as the new components satisfy the QoS specification.

We also need QoS specifications to monitor the satisfaction of QoS requirements at runtime. Compliance monitoring allows a system to adapt to its runtime environment, and it allows a QoS management system to diagnose which components are at fault if a system does not satisfy the user-level QoS requirements.

We want to facilitate services that can provide different modes of QoS, for example, high availability with medium performance, or medium availability with high performance. By offering different QoS modes, services can allow their clients to make appropriate trade-offs. Not only do multi-mode services require QoS specifications to characterize various modes, they also require multi-category specifications that incorporate different QoS categories, such as performance, availability, security, and timing in the same QoS specification.

In open distributed systems, clients and services can be added and removed dynamically. Moreover, no single entity is in control of system evolution. When deploying a client in an open system, we may not know which services it will use over time. Nevertheless, the client may still have certain QoS requirements. We need to specify these requirements and use them as part of the client-server binding process to ensure that clients bind to services that satisfy their QoS requirements. The binding process could be accomplished by a trader service [Object Management Group 1997a] that matches QoS requirements of clients with QoS properties of services, or it may involve more elaborate negotiation protocols between clients and services to arrive at a QoS agreement. To enable QoS-based design, trading and negotiation, we need a language to specify QoS requirements. Moreover, we need to communicate QoS specifications as first class values between clients, services, and traders.

To support the specification of QoS properties, we introduce a language called

QML (QoS Modeling Language). We use a simple example to illustrate the use of QML for QoS specification.

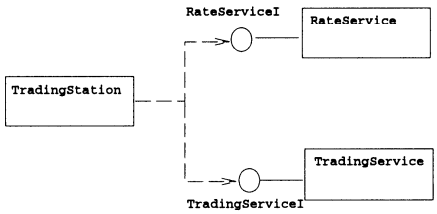

Fig. 1. UML class diagram for the currency trading system

Consider the currency trading system in Figure 1. We use UML [Booch et al. 1997] to depict the system structure. Human 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. Since an inaccessible currency trading system might incur significant financial loss, it is essential that the system be highly available. It must also peform adequately enough to provide recent information.

Consider the CORBA IDL [Object Management Group 1995] 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.

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

Fig. 2. The `RateServiceI` interface

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 `systemReliabilty` of type `Reliability`. The 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 entities 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, unless stated otherwise. 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 a 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 that must be imposed on individual components to enable the system as a whole to meet its QoS goals. QoS specifications can also be negotiated dynamically between clients and servers in distributed systems. Such negotiations can be performed either through a QoS-enabled trader service [Object Management Group 1997a] or by using advanced QoS negotiation protocols. If we use a trader service, ervices may register themselves and their QoS characteristics with that trader service. This enables clients to search for matching services that supports a specific interface with a minimal level of QoS. It is essential in such searches to have the ability to determine whether the QoS characteristics of the advertised service in fact satisfy the characteristics required by the client. As an example, satisfying the constraint "delay $<$ 10 msec" implies that we also satisfy "delay $<$ 20 msec". We call this procedure *conformance checking*, which is supported by QML.

Once we have QoS agreement we must be able to monitor the adherence to such deals. Thus, we anticipate that QoS specifications will also be used as input

```
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
  };
};
```

Fig. 3. Contracts and Profile for RateServiceI

for monitoring and charging mechanisms. Due to the vast number of ways QoS specifications might used, we consider the agreement and binding to an agreed upon QoS specification outside the scope of the specification language. This paper focuses on how to specify QoS, not on how to use QoS specifications.

QML specifications are similar to traditional interface specifications in the sense that they *describe* behavior—they cannot be executed to implement behavior. In fact, we can consider QML specifications as abstraction boundaries that extend the traditional notion of interface to also cover QoS properties. Notice, however, that the association between functional interfaces and a QML specification is flexible and possibly dynamic.

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. QML captures the fundamental concepts involved in the specification of QoS properties. Although special-purpose QoS languages do exist, there has been a lack of general-purpose languages that allow engineers and users to express the wide variety of QoS aspects used in open distributed systems.

QML is designed to be a general QoS specification language for the development of distributed object-oriented systems. We want QML to integrate seamlessly with existing object-oriented concepts. This overall goal results in the following specific 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 could have two QoS specifications: a specification that captures the client's requirements and a specification that captures the service's provisioning. This separation allows us to specify the QoS characteristics of a component and, the QoS properties that it provides and requires, without specifying the interconnection of components. This 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 requirements of a client. These requirements are a consequence of the separate specification of both 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. QML must allow QoS specifications for interfaces, operations, attributes, operation parameters, and operation results.

To specify QoS one needs to identify an appropriate vocabulary—dimensions. QML allows user-defined dimensions. If these dimensions are independent of their underlying mechanisms, QML allows the specification of QoS properties independently of how these properties are implemented, for example, the QML specification of a certain level of availability without reference to a particular high-availability mechanism, such as primary-backup or active replication. Having a language to specify QoS properties abstractly gives a clean separation between deciding which properties should be provided and implementing these properties.

We organize the rest of this paper in the following way. In Section 2, we introduce the fundamental concepts that QML supports. The purpose of Section 2 is to provide an intuitive description of QoS specifications in a syntax-neutral way. In Section 3, we introduce the syntax for QML and describe its semantics precisely, but informally. To illustrate the use of QML in practice, we show how to use it at design time in the context of UML [Booch et al. 1997]. We introduce a set of UML extensions to support QML in Section 4. We show how to use these extensions in particular and QML in general, by describing the design of a computer-based telephony using QoS specifications. We present the formal semantics for QML in Section 6. The topic of Section 7 is related work, and, finally, in Section 8 we draw our conclusions.

2. FUNDAMENTAL CONCEPTS

We characterize QoS along named *dimensions*, such as latency, throughput, failure semantics, and encryption level. Moreover, we group related dimensions into *categories*. For example, we could group the dimensions latency and throughput into a category called performance. In Section 2.1, we describe the fundamental concepts underlying our notion of QoS dimensions. In Section 2.2, we outline the mechanisms we use to create new abstractions that contain dimensions.

2.1 Dimensions

A dimension consists of a name and a domain of values. A QoS specification along a given dimension is a constraint over the dimension's domain. For example, we could have a dimension called latency, which captures the time it takes for a service to reply to method calls. The domain for this dimension would then be values that capture elapsed time. The non-negative real numbers would be one possible domain for latency. A QoS specification along the latency dimension could be that the latency is less than 10 seconds, which could be expressed by the constraint "latency $<$ 10 seconds." This is a constraint over the non-negative real numbers. The constraint is satisfied by a subset of the domain, namely the non-negative real numbers that are strictly less than 10. The domain that we use for numeric constraints is called the *numeric domain*. It contains the real numbers.

It is not always meaningful to use numeric domains. For example, assume that we want to specify QoS along a dimension, called failure semantics, that characterizes how services behave after a failure. We can identify a number of different behaviors, such as halt, initial state, and roll back. Halt means that the server stays down, initial state means that the server comes up in its initial state; and roll back means that the server comes up in some previous, consistent state. A particular service will behave in one of these three ways. A QoS specification could point out one of

these behaviors as in "failure semantics == halt." The domain used for the failure semantics dimension is called an *enumerated domain*. Its elements are user-defined values specified as names, such as halt.

For some dimensions, we want domain elements to be sets of names rather than names. For example, consider a dimension, called failure masking, that characterizes the types of failures that a service exposes to its clients. We can introduce names that capture the various types of possible failures. Possible candidate names are lost reply, lost request, and invalid reference. A particular service will then expose a subset of these possible failure types. Thus, a QoS specification along the failure masking dimension could point out a set of names as in "failure masking == { lost reply, lost request }." A domain whose elements are sets of names is called a *set domain*.

It is often the case that one domain element reflects a higher level of service than another domain element. For example, for the failure semantics dimension, a service that comes back up provides a higher level of service than a service that stays down. Thus, the names initial state and roll back capture a higher level of service than halt. In another example, for the latency dimension, the domain element 8 captures a higher level of service than the domain element 10 because it corresponds to a smaller response time. We want to capture this notion of better than or stronger than for domain elements.

The first step towards a stronger than relation on domains is an ordering relation on domains. The numeric domain already comes with a "built-in" ordering "<." For enumerated and set domains, we need to specify a user-defined ordering over the domain elements. Once we have an ordering, we also need to specify if larger elements are stronger than smaller elements or whether smaller elements are stronger than larger elements. For example, for latency smaller numbers are stronger than larger numbers, but for throughput larger numbers are stronger than smaller numbers. We say that a domain is *increasing*, if larger elements are better, and we say that a domain is *decreasing*, if smaller elements are better.

For an enumerated domain, an order specified for the names will also describe an order for the domain elements, since the names are the domain elements. This is not the case for set domains. For set domains, the elements are sets of names. For a set domain, we specify an ordering on the names used, and we use this name ordering, in conjunction with subset inclusion, to define an ordering on the sets of names.

If we specify an ordering for a set or enumerated domain, we can also express inequality constraints over that domain. For example, we could write a constraint like "failure semantics > halt." This constraint is satisfied by either initial state or roll back. For a domain that does not have an ordering, we can only express equality constraints.

We use the stronger than relation on domains to define the conformance relation for QoS specifications. Intuitively, a "stronger" QoS specification conforms to a "weaker" QoS specification. We use conformance to check the legality of client-server connections: the specification that captures the server QoS properties must conform to the specification that captures the client QoS requirements. We do not want to insist that the client and server specifications be equal; they may have been written by different people at different times. The server may provide stronger QoS

properties than that required by the client. We also use conformance to provide substitutability of service implementations. We can substitute an old service implementation with a new service implementation as long as the QoS properties of the new implementation conform to the QoS properties of the old implementation.

We also use the term *conformance* for domain elements. We say that a domain element conforms to another domain element if that former element is stronger than the latter.

In the above discussion, we have described only dimensions that have a single constraint. In some cases, we need to allow a dimension to specify multiple constraints. Multi-constraint dimensions are necessary when a dimension can capture a QoS property that varies over time and when we want to express statistical constraints over this variation. For example, the latency dimension captures a property that varies over time: the response time for calling a method in a service. Rather than expressing a single constraint, such as "latency < 10," we may want to express constraints over the variance and mean for response times. We want to express constraints of the form "latency { **variance** $< .2$; **mean** < 7 }." We consider variance and mean to be *aspects* of a dimension. We introduce aspects such as average, mean, percentile, and frequency.

2.2 Abstraction Mechanisms

The goal of this section is to provide an overview of the abstraction mechanisms that we use to construct QoS specifications in QML.

A *contract type* represents a QoS category such as performance, availability, security, or timing. A contract type describes the dimensions of a category, it specifies the name, domain, and possibly a user-defined ordering for each dimension. For example, we could define a contract type, called performance, that describes a latency and throughput dimension. These dimensions would have numeric domains, and the latency dimension would be specified as decreasing whereas the throughput dimension would be specified as increasing.

A *contract* is an instance of a contract type and represents a particular QoS specification within a given category. For example, a performance contract could specify particular constraints for the throughput and latency dimensions defined in the performance contract type. A contract aggregates a number of constraints.

We use contracts to capture QoS specifications for interface elements, such as operations, attributes, operation parameters, and operation results. An interface element will typically have multiple contracts: one for each QoS category of interest. For example, the same operation may have a contract that specifies its performance properties and a contract that specifies its availability properties.

A *profile* describes the associations between contracts and interface elements for a particular interface. A profile provides an aggregation mechanism for contracts. Where contracts are used for interface elements, profiles are used for interfaces. Since different implementations may have different QoS properties, we can define different profiles for the same interface. As a notational convenience, a profile can specify a default contract, which applies to all interface elements. In addition, the profile can associate more stringent contracts with individual interface elements. As an example, we could associate a default performance contract with the operations of an interface and then add stronger contracts for operations that are required to

have better performance than that stated by the default contract.

A contract can be specified as an incremental refinement of another contract. The result of refinement is a QoS contract with more stringent QoS properties. For example, we could define a performance contract that specifies a latency of 10 milliseconds as a refinement of a performance contract that specifies a latency of 20 milliseconds. Refinement is a notational convenience for the common case in which one contract is derived from another contract. A contract can only refine another contract of the same type. Since we insist that refinement results in a stronger contract, refinement implies conformance.

Profile refinement is defined in terms of contract refinement. Essentially, a profile is refined by refining its contracts. As for contracts, profile refinement also results in a profile with more stringent QoS properties.

Refinement is a statically defined relationship between two profiles, much like inheritance in most object-oriented languages. Sometimes, we need to be able to determine if two statically unrelated profiles can be ordered. More specifically, we are interested in whether one profile imposes stronger QoS requirements that another. If so, we can conclude that a service providing the former can in fact replace a service providing the latter. If a profile P is stronger than a profile Q we say that P *conforms* to Q. QML introduces a definition of when a profile conforms to another if the two profiles are associated with the same interface.

3. LANGUAGE CONCEPTS

In the following sections we introduce an abstract syntactic description and additional language detail for QML. Section 6 presents a more precise semantic description.

3.1 Contracts and Contract Types

To capture the structure of contracts within a given QoS category C, a contract type specifies a *dimension type* for each dimension within C. The dimension type for a dimension D determines how individual contracts specify the QoS properties along the D dimension. Figure 4 gives an abstract syntax for contract types, and Figure 5 gives an abstract syntax for individual contracts that are instances of contract types.

A dimension type specifies a domain for the dimension; QoS properties are specified as constraints over this domain. The domain may be ordered, and it may have an associated unit. We use three different domain types: set, enumeration, and numeric. Set and enumeration domains are user-defined domains; the numeric domain is a built-in domain. For example, consider a dimension called *latency* that captures response time. The domain for latency would be numeric, and constraints would be of the form "latency < 15." As illustrated by this example, a constraint has an operator "<" and a domain element "15". In general, we use the following set of operators: "{==, <, <=, >, >=}." We allow only inequality operators for ordered domains.

In the syntax description we use italics for production names such as *conType*. Keywords are written in **boldface** while other terminal symbols are <u>underlined</u>. We use dots (...) to indicate a sequence of structurally similar entities. We assume the existence of a set Name . The elements in Name are the names of dimensions or

$$
\begin{array}{lcl}
\textit{conType} & ::= & \textbf{contract}\ \{\textit{dimName}_1 : \textit{dimType}_1\ ; \ldots\ ;\ \textit{dimName}_k : \textit{dimType}_k ;\} \\
\textit{dimName} & ::= & n \\
\textit{dimType} & ::= & \textit{dimSort} \\
 & | & \textit{dimSort unit} \\
\textit{dimSort} & ::= & \textbf{enum}\ \{n_1 , \ldots , n_k\} \\
 & | & \textit{relSem}\ \textbf{enum}\ \{n_1 , \ldots , n_k\}\ \textbf{with}\ \textit{order} \\
 & | & \textbf{set}\ \{n_1 , \ldots , n_k\} \\
 & | & \textit{relSem}\ \textbf{set}\ \{n_1 , \ldots , n_k\} \\
 & | & \textit{relSem}\ \textbf{set}\ \{n_1 , \ldots , n_k\}\ \textbf{with}\ \textit{order} \\
 & | & \textit{relSem}\ \textbf{numeric} \\
\textit{order} & ::= & \textbf{order}\ \{n_i < n_j , \ldots , n_k < n_m\} \\
\textit{unit} & ::= & \textit{unit/unit} \mid \% \mid \textit{msec} \mid \ldots \\
\textit{relSem} & ::= & \textbf{decreasing} \mid \textbf{increasing}
\end{array}
$$

Fig. 4. Abstract syntax for contract types

$$
\begin{array}{lcl}
\textit{contract} & ::= & \textbf{contract}\ \{\textit{constraint}_1\ ; \ldots\ ;\ \textit{constraint}_k ;\} \\
\textit{constraint} & ::= & \textit{dimName constraintOp dimValue} \\
 & | & \textit{dimName}\ \{\textit{aspect}_1\ ; \ldots\ ;\ \textit{aspect}_n ;\} \\
\textit{dimValue} & ::= & \textit{literal unit} \\
 & | & \textit{literal} \\
\textit{literal} & ::= & n \\
 & | & \{n_1 , \ldots , n_k\} \\
 & | & \textit{number} \\
\textit{aspect} & ::= & \textbf{percentile}\ \textit{percentNum constraintOp dimValue} \\
 & | & \textbf{mean}\ \textit{constraintOp dimValue} \\
 & | & \textbf{variance}\ \textit{constraintOp dimValue} \\
 & | & \textbf{frequency}\ \textit{freqRange constraintOp number\%} \\
\textit{freqRange} & ::= & \textit{dimValue} \\
 & | & \textit{lRangeLimit dimValue , dimValue rRangeLimit} \\
\textit{lRangeLimit} & ::= & (\ |\ [\\
\textit{rRangeLimit} & ::= &)\ |\ ] \\
\textit{constraintOp} & ::= & ==\ |\ >=\ |\ <=\ |\ <\ |\ > \\
\textit{percentNum} & ::= & 0 \mid 1 \mid \ldots \mid 99 \mid 100 \\
\textit{dimName} & ::= & \underline{\textit{defined in Figure 4}} \\
\textit{unit} & ::= & \underline{\textit{defined in Figure 4}}
\end{array}
$$

Fig. 5. Abstract syntax for contracts

the names of elements used in set and enumeration domains. We use n and *name* to refer to an element of Name. The *number* production represent regular numeric literals. A more strictly define syntax can be found in appendix A where we use Extended Backus-Naur Form.

3.1.1 *Domains, Orders, and Units.* We specify set domains with the following syntax: "**set** $\{n_1 , \ldots , n_k\}$." The names "$n_1 , \ldots , n_k$" are the names of the domain. The domain contains all possible subsets of these names. Each element in the domain is a subset of names; each domain element captures a possible service behavior. For example, each name could reflect a possible failure symptom that services in general may expose to their clients. The domain would then contain sets

of failure symptoms. We can use a particular set of failure symptoms (a domain element) to characterize the failure masking behavior of a particular service.

We use "**enum** $\{n_1, \ldots, n_k\}$" to specify an enumeration domain. The names "$\{n_1, \ldots, n_k\}$" are the domain elements. Each name captures a possible service behavior. As an example, we can use an enumeration to specify QoS properties along a *data policy* dimension. There are two possible data policies: either data is always valid following a failure or data is always invalid following a failure. We can specify the possible policies as the enumeration **enum** $\{$*valid*, *invalid*$\}$.

The numeric domain contains the real numbers. We specify QoS properties along a numeric dimension as numeric constraints—constraints for which the value is a number. An example of a numeric dimension is availability. The availability of a service is specified as the probability that the service is responding to requests (i.e., not down). In reliability contracts, we would specify 99% availability as *availability* $>=$ 99%.

The numeric domain comes with a built-in total order. For set and enumeration domains, we need to specify ordering explicitly. We describe ordering of names in the following way: "**with order** $\{n_i < n_j, \ldots, n_k < n_m\}$." Thus, an ordering is specified as a set of pairs, where each pair defines the relative order between two names. The defined ordering is transitive. This means that if we have asserted an ordering $\{a < b, b < c\}$, we also assert that a and c are mutually ordered.

The order clause specifies only an order on names. For set domains, we need to extend this order to apply to sets of names. If A and B are subsets in the same set domain, we extend the name ordering in the following way:

$$A < B \Leftrightarrow \forall a \in (A \setminus B) : \exists b \in (B \setminus A) : a < b$$

Here, a and b are names, and the expression "$a < b$" refers to the ordering defined for names.

The ordering on sets amounts to set inclusion if the names are not ordered. If the names are ordered, A does not have to be included in B, but the names in A that are not in B must be smaller than at least one name in B.

A dimension declaration can be given a unit. The unit can be simple, such as *seconds* or *percent* (%), or it can be a combined unit such as *failures/hour*. If a unit is specified in a contract type, QML requires that all instances of that contract type use the same unit.

3.1.2 *Conformance.* We want to define a conformance relation for contracts. To define conformance for contracts, we first need to define conformance for the various types of constraints that can be written as a part of contracts. We define conformance for constraints that are defined for the same dimension.

For a set dimension we want 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

```
contract {
  s1 : decreasing set { e1, e2, e3, e4 } with order {e2<e1, e1<e3, e3<e4};
  s2 : increasing set { f1, f2, f3 };
  e1 : increasing enum { a1, a2, a3 } with order {a2<a1, a3<a2};
  e2 : enum { b1, b2, b3 };
  n1 : decreasing numeric msec;
  n2 : increasing numeric mb / sec;
};
```

Fig. 6. Example contract type expressions

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 we 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, if we have a decreasing dimension d we want the constraint $d < 10$ to conform to the constraint $d < 20$. Since the values that satisfy the first constraint also satisfy the second constraint, we want to consider the first constraint stronger than the second. 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.

The desire to have conformance correspond to constraint satisfaction is based on the assumption that most programmers would consider it counter intuitive for $d < 5$ to not conform to $d < 3$. Thus, we regard those situations as errors.

The example in Figure 6 gives a few examples of how dimensions can be declared in a contract type. $s1$ is a *partially ordered decreasing* set. This means that smaller sets are better and that only some of the names in the set can be related. The second set is called $s2$ and is an unordered *increasing* set.

$e1$ and $e2$ are both enumerated dimensions, but only $e1$ has a partial order defined. Finally, we have two numeric dimensions. For $n1$, smaller values are considered stronger while for $n2$ larger values are consider stronger. Numeric dimensions are always totally ordered.

A contract may omit specifying QoS properties along a particular dimension listed

in its contract type. Omission of a specification indicates that the QoS property along that dimension is undefined and may take any value with any distribution with the specified domain.

3.1.3 *Aspects.* As indicated by Figure 5, the general way of specifying a constraint for a dimension is by defining an *aspect*. An aspect can be a statistical characterization such as a percentile or mean. It can also be a contract-type specific, user-defined characterization previously unknown in the QML language. QML currently includes four generally applicable aspects: *percentile*, *mean*, *variance*, and *frequency*.

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. As an example, assume that we are measuring delay and that we have obtained the following ten measurements: $2, 3, 3, 4, 4, 4, 5, 6, 6, 8$. The expression "**percentile** 60" would denote the six lowest values, since these are considered the strongest for delays. The expression "**percentile** $80 < 6$" is not satisfied by the measurements above, but the constraint "**percentile** $100 <= 8$" is. We can use percentiles to express constraints that restrict the proportions of different ranges of values. We allow a set of aspects for a dimension to contain more than one percentile constraint, as long as the same percentile P does not occur more than once. QML implementations are not required to check for inconsistencies among the constraints; thus, consistency is solely the responsibility of the specifier. We only allow percentile characterizations for dimensions with numerical domains.

Sometimes, there is a need to specify the frequency of individual values or values in certain ranges. QML 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. If a frequency is defined for a range rather than a specific value, we use parenthesis and square brackets for open and closed boundaries respectively. If we use an open boundary, the boundary value is included, while closed boundaries exclude the boundary value. The values included in a range (a, b) with open boundaries are those larger than or equal to a and smaller than or equal to b. I a boundary is closed the boundary value itself in not included in the 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.

The mean and variance aspects are used to define the mean and variance, respectively, of measured values over some time period. As for simple dimensions, mean can be used to define a constraint only for the weakest acceptable value. Thus, for increasing numeric dimensions we can define a lower boundary for mean, but we can not define an upper bound. Likewise, for a decreasing numeric dimension we may define only an upper bound for mean.

The variance is defined as the expected value of a random variable (the dimension)

which is subtracted from its mean and squared. The variance is a positive number that indicates how much occurrences may vary from the mean. We allow only inequalities that specify upper bounds for variance.

Not all of the allowed statistical characterizations make sense for every dimension. In general, we need to understand the scale type [Fenton 1991] before we know whether a particular statistical method can be applied to a dimension. The scale type is determined by the dimension characteristics defined in a contract type and the semantics of the dimension. By semantics, we mean how dimension values are interpreted by functions such as a monitoring function.

To make a statistical characterization with a known confidence we need a sufficiently large sample. This aspect of QoS characterization must be taken care of by the components that use QML specifications. An example of such a component is the monitoring component that measures delivered QoS at runtime.

Figure 7 shows some examples of constraints in contract expressions. The contract expression is preceded by the name of its corresponding contract type. We show how to define contract types in Section 3.2. The first example shows simple constraints. The constraint on `s1` illustrates our use of comparison for set values. In this case we require that actual `s1` values are stronger than or equal to the set literal value {`e1,e2`}. For `s2` we require that the actual value is equal to { `f1` }. The remaining constraints are for enumerated and numeric dimensions.

The second contract expression in Figure 7 is more complicated. 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 `n2` defines bounds for values in different percentiles over the measurements of `n2`. In addition, we define an upper bound for the mean and the variance.

3.2 Definition of Contracts and Contract Types

Here we introduce a syntax for defining contracts and contract types. The definition of a contract type binds a contract type to a variable; the definition of a contract binds the result of a contract expression to a variable. Figure 8 gives an abstract syntax for contract and contract type definition.

In the syntax, we assume the existence of a set Var of variables. We use x_c, x_p, and y as typical elements of Var ($x_c, x_p, y \in \mathsf{Var}$). x_c denotes a variable that holds contract values, x_p denotes a variable that holds profile values, and y denotes a variable that holds contract types.

We use the syntax "**type** `Performance` = **contract** {...}" to define a contract type called `Performance`. When we define a contract, we explicitly specify the type of the contract. The definition "`traPerf` = `Performance` **contract**{...}" defines a performance contract with the name `traPerf`. This definition requires that the variable `Performance` has previously been bound to a contract type in a definition of the form "**type** `Performance` = ..."

We use explicit typing to allow different contract types to have similar dimension names. For example, assume that two types T_1 and T_2 both have a dimension

```
contractTypeName contract {
  s1 <= { e1, e2 };
  s2 ==  {f1 };
  e1 < a2;
  e2 == b2;
  n2 < 23;
  n3 > 45;
};

contractTypeName contract {
  s1 { percentile 20 < { e1, e2 }};
  e1 {
        frequency a1 <= 10 %;
        frequency a2 >= 80 %;
        frequency a3 < 20 %;
        frequency a3 >= 5 %;
      };
  e2 == b2;
  n2 {
        percentile 10 < 20;
        percentile 50 < 45;
        percentile 90 < 85;
        percentile 100 <= 120;
        mean >= 60;
        variance < 0.6;
      };
  n3 {
        mean > 100;
        variance < 0.4;
      };
};
```

Fig. 7. Example contract expression

decl	::=	*conTypeDecl* | *conDecl*
conTypeDecl	::=	**type** y = *conType*
conDecl	::=	x_c = *conExp*
conExp	::=	y *contract*
		| x_c **refined by** $\{constraint_1 ; \ldots ; constraint_k ;\}$
conType	::=	*defined in Figure 4*
contract	::=	*defined in Figure 5*
constraint	::=	*defined in Figure 5*

Fig. 8. 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 };
  numberOfFailures : decreasing numeric failures/year;
  MTTR : decreasing numeric sec;
  MTTF : increasing numeric day;
  reliability : increasing numeric;
  availability : increasing numeric;
};
```

Fig. 9. Example contract type definition

called D. If a contract C specifies only a QoS property along D and omits the specification of other dimensions, we need explicit typing to determine if C is of type T_1 or T_2.

We can define a contract A to be a refinement of another contract B using the construct "$A = B$ **refined by** $\{\ldots\}$" where B is the name of a previously defined contract. The contract that is enclosed by curly brackets ($\{\ldots\}$) 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. For a refinement to be semantically valid we require that the result of the refinement, A, is more stringent than the contract being refined, B. For example, the delta can specify QoS properties along dimensions for which specifications were omitted in B. Moreover, the delta can strengthen the QoS properties specified in B. We discuss contract refinement in more detail in Section 3.5.

To illustrate our definition syntax and informally describe its semantics, we give an example of a contract type in Figure 9 and examples of contracts in Figure 10. The contract type `Reliability` has the dimensions for reliability identified in [Koistinen 1997].

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` provides only a guarantee about the invocation semantics; it does not provide guarantees for the other dimensions in the `Reliability` contract type.

In Figure 10, the contract `nameServerReliability` is defined as a refinement of another contract, namely the contract bound to the name `systemReliability`. In this example, we strengthen the `systemReliability` contract by providing a specification along the `serverFailure` dimension, which was left unspecified in the `systemReliability` contract.

```
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 };
};
```

Fig. 10. Example contract definitions

profile	::=	**profile** $\{req_1 ; \ldots ; req_n ;\}$
req	::=	**require** *contractList*
	|	**from** *entityList* **require** *contractList*
contractList	::=	$conExp_1 , \ldots , conExp_n$
entityList	::=	$entity_1 , \ldots , entity_n$
entity	::=	*opName*
	|	*attrName*
	|	*opName.parName*
	|	**result of** *opName*
opName	::=	*identifier*
attrName	::=	*identifier*
parName	::=	*identifier*
conExp	::=	*defined in Figure 8*

Fig. 11. Abstract syntax for profiles

3.3 Profiles

A service specification contains an interface and a QoS profile. The interface describes the operations and attributes exported by the service; the profile describes the QoS properties of the service. A profile is defined relative to a specific interface and 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 have multiple implementations with different QoS properties.

Here we describe a syntax for profile values, and in Section 3.4 we describe a syntax for profile definition. Figure 11 gives an abstract syntax for profiles. A profile is a list of requirements, each of which specifies one or more contracts for one or more interface entities. An interface entity is an operation, an attribute, an operation parameter, or an operation result. If a requirement is stated without an associated entity, the requirement is a *default requirement* that applies to all

```
declaration   ::= conTypeDecl
                | conDecl
                | profileDecl
profileDecl   ::= x_p for intName = profileExp
profileExp    ::= profile
                | x_p refined by {req_1 ; ... ; req_n ;}
intName       ::= identifier
conTypeDecl   ::= defined in Figure 8
conDecl       ::= defined in Figure 8
profile       ::= defined in Figure 11
req           ::= defined in Figure 11
```

Fig. 12. Abstract syntax for definition of profiles

entities within the interface in question. Our intention is that the default contract be the strongest contract and that it apply to *all* entities within the interface. We can then use refinement to explicitly specify a stronger contract for individually selected entities.

Contracts for individual entities are defined by introducing a delta that strengthens the default contract by the profile. This way we avoid introducing new named contracts for each refinement within a profile.

A profile may contain at most one default contract of a given contract type. That is, we cannot specify multiple default contracts for performance, but we can omit the specification of a default contract. Similarly, at most one contract of a given type can be explicitly associated with an individual 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, no QoS properties within the category of T are associated with that entity. In the syntax definition below x_p denotes profile names.

3.4 Definition of Profiles

A profile definition binds the result of a profile expression to a variable. A profile definition is given relative to a particular interface; the profile can only be used in conjunction with that interface and with the interfaces that inherit from the specified interface. The syntax for profile definition is given in Figure 12. The definition "x_p **for** *intName* = *profileExp*" binds the variable x_p to the profile that is the result of evaluating the profile expression *profileExp* with respect to the interface *intName*.

A profile expression (*profileExp*) can be a profile or a variable with a "{...}" clause. If a profile expression contains a variable with a "{...}" clause, the variable must be bound to a profile, and the "{...}" clause is then a delta that refines this profile. The definition gives a name to this refined profile and associates it with the interface. We discuss the semantics for profile refinement in Section 3.5.2.

A general requirement is that the interface entities referred to by the profile must exist in the related interface.

To exemplify the notion of profile definition, consider the interface of a name server in Figure 13. The profile called `nameServerProfile` is a profile for the

```
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;
  };
}
```

Fig. 13. The interface of a name server

`NameServer` interface; it associates various contracts with the operations defined with the `NameServer` interface. The `nameServerProfile` defines the contract called `nameServerReliability` (introduced in Figure 10) to be 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 operations must always refine the default contract, it is implicitly understood that the contract expression in an operation contract is, in fact, a refinement.

3.5 Refinement

There we give an informal semantics for profile and contract refinement; Section 6 contains a formal definition of refinement semantics.

The main motivation for profile refinement is that, in an object-oriented setting, interfaces are typically subject to sub-typing or inheritance relationships. Since interfaces can be defined through derivation and since there is a close coupling between profiles and interfaces, we want to enable derivation of profiles; we do not want to insist that all profiles be defined from scratch.

Refinement results in a stronger profile. The reason for this semantic property is that we want to maintain sub-type substitutability. Most object-oriented languages support a notion of sub-type substitutability for functional interfaces. We want to support a similar notion in the area of QoS specifications. Although it is possible to define two independent profiles so that one is stronger than the other, refinement provides a more disciplined way to achieve this.

Profile refinement is defined in terms of contract refinement. Another motivation for contract refinement is to support the notion of a default contract within a profile and to allow stronger contracts to be specified for selected interface entities.

3.5.1 *Contract Refinement.* There are two issues in contract refinement: which contract A can refine a contract B and what is the result of a contract A refining contract B. The first issue is concerned with legality of refinement, and the second issue is concerned with the result of the refinement given that the refinement is

```
A = Reliability contract {
  failureMasking <= { omission, response };
  dataPolicy == valid;
  availability > 99.9 %;
};

B = A refined by {
  failureMasking <= { omission }; //legal
  dataPolicy == invalid; //illegal
  availability > 98 %; //illegal
  rebindPolicy == rebind; //legal
};
```

Fig. 14. Examples of legal and illegal contract refinement

semantically legal.

The first criterion for legality is that the contracts A and B must be of the same type; thus, a reliability contract cannot refine a performance contract. If the contracts are of the same type, we need to examine each constraint in both contracts.

The intuition behind contract refinement is that A can add new constraints, replacing the constraints in B with stronger constraints. More precisely, we can capture refinement by the following rules:

—If only one of the two contracts specifies a constraint along a given dimension, refinement is legal along that dimension, and the result of the refinement is the single constraint specified.

—If both A and B specify a constraint along a given dimension, the constraint in A must conform to (be stronger than) the constraint in B, and the result of refinement is the constraint in A.

The examples in Figure 14 illustrate our rules for contract refinement. In the figure, the contract `A` is refined, and the result of the refinement is bound to `B`. Refinement along the `failureMasking` dimension is legal because the delta contains a stronger constraint: the singleton set {`omission`} is stronger than the set {`omission`, `response`} because the domain is decreasing. Refinement along the `dataPolicy` dimension is illegal. We cannot strengthen a constraint which uses an equality operator. The `availability` dimension is also refined illegally because the constraint in the delta is not stronger than the constraint in `A`. Finally, it is legal to add a constraint for the `rebindPolicy` dimension.

3.5.2 *Profile Refinement.* We first illustrate the profile refinement for a specific example, and then we give precise, but informal, rules for the general case. In Section 6, we give a formal definition of the rules for profile refinement.

Figure 15 shows an example of profile refinement. The profile `P2` is described as a refinement of a profile `P1`. `P1` is refined by a profile delta that specifies the difference between `P1` and `P2`. The interface `I2` in turn is derived from the interface `I1`. `E` is an entity in `I1` and due to interface inheritance also an entity in `I2`.

The contract for `E` in `P1` is the default contract `D1` refined by the individual

```
P1 for I1 = profile {
  require D1;
  from E require C1;
};

P2 for I2 = P1 refined by {
  require D2;
  from E require C2;
};
```

Fig. 15. Example of profile refinement

contract C1. The contract for E in P2 is the result of several refinement operations: it is defined as D1 refined by C1, D2, and C2.

To describe the profile refinement rules for the general case, we need to clarify some terminology. The *specified* contract is the literal contract that is specified as part of a requirements clause. For example, C1 is the specified individual contract for E in P1, and D2 is the specified default contract in P2. An *effective* contract is a contract that applies to a profile after profile refinement has been performed. For example, the effective individual contract for E in P2 is C1 refined by C2, and the effective default contract for P2 is D1 refined by D2. For a profile that is not specified using refinement, the effective and specified contracts are the same.

In the general case, a profile A refines a profile B, and the result of the refinement is a profile C. The profile B is defined for an interface I_B, and the profile C is defined for an interface I_C. The profile A is the refinement delta. We define the rules for profile refinement in terms of A, B, and C. The goal is to specify, in general terms, which contracts apply to an entity E in the profile C. We can specify this in terms of effective default and effective individual contracts as follows:

—The contract for E in C is the effective default contract for C, if any, refined by the effective individual contract for E in C, if any.

Notice that this rule is relative to a specific contract type T, so, for example, when we say *the default contract*, we mean *the default contract of type* T. Notice also that some of the contracts may be "missing," for example, there may not be a default contract. A refinement operation without a contract has no effect. If both contracts are missing, the result is an "empty" contract.

For this definition to be complete, we need to give the general rules to determine the effective individual and effective default contracts:

—The effective default contract in C is the effective default contract in B, if any, refined by the specified default contract in A, if any.

—The effective individual contract for E in C is the effective individual contract for E in B, if any, refined by the specified individual contract for E in A.

It may not always be legal for a profile A to refine a profile B. For profile refinement to be legal, all the contract refinements involved must be legal.

Consider the definitions in Figure 16. Both the `nameServerProfile` and the `maskingProfile` apply to the `NameServer` interface. The profile `maskingProfile` attempts to refine the `nameServerProfile`. The profile refinement is illegal: the

```
nameServerReliability = Reliability contract {
  operationSemantics == once;
  serverFailure == rolledBack;
};

masking = Reliability contract {
  failureMasking == { response, stateTransition };
  serverFailure == halt;
};

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

maskingProfile for NameServer =
  nameServerProfile {
    require masking;  // Illegal
    from lookup require Reliability contract { dataPolicy == valid;};
  };
```

Fig. 16. Example profile refinement

default contract, `masking`, in the delta, does not refine the default contract in `nameServerProfile` because they both specify an equality constraint for the dimension called `serverFailure`, and the values are not equal.

3.6 Profile Conformance

Conformance is different from refinement in that conformance allows us to compare two syntactically unrelated profiles. We say that a profile P conforms to another profile Q if satisfying P also implies satisfying Q. We can only determine conformance for profiles associated with the same interface.

There is an obvious relation between refinement and conformance. The relation is that if a profile P' is a refinement of another profile P, P' will also conform to P. Conformance is, however, more general and can be applied to any pair of profiles as long as they are profiles of the same interface.

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—which we also call the *stronger than* relation—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. The notion of constraint conformance is treated informally in the next section and more formally in section 6.6.

aspect	*conformance rule*
frequency	For every constraint of type **frequency** $R * P$ in D there must be a constraint with the same aspect signature in S that is stronger or equally strong with respect to the value P.
percentile	For every constraint of type **percentile** $P * V$ in D there must be a constraint with the same aspect signature in S that is stronger or equally strong with respect to the value V.
mean	If D has an mean constraint, the mean constraint for S must be stronger or equal to the one in D.
variance	If D has a variance constraint, the variance constraint of S must be smaller or equal to the variance of D.

Fig. 17. Conformance rules for aspects

3.6.1 *Comparing Dimension Constraints.* For refinement as well as conformance we need a notion of a dimension constraint being stronger than another dimension constraint. In this section we will informally define the rules that we use in QML to determine when a constraint is stronger that another constraint. Although this relation is used in both contract refinement and conformance, we call the relation dimension *conformance*. We will sometimes say that a stronger value *conforms* to a weaker value, or simply that one constraint is stronger than another.

Earlier in this paper we stated that a dimension has a domain, which may have an ordering and a relation semantics declaration. The ordering defines how the elements of the domain are ordered, and relation semantics defines how the *stronger than* relation is mapped to the ordering. The mapping differs slightly for the three basic domain types that QML allows. As an example, for an *increasing* numeric dimension, the value 3 is considered stronger than 1. For a *decreasing* numeric dimension it is the other way around. For sets, however, *increasing* means both that larger sets are stronger and that larger element values are stronger.

In a QML description we can use various operators to express constraints. The set of allowed operators is different depending on how the *stronger than* relation is mapped to the ordering. For example, we do not allow the $<$ operator for an increasing domain. In the following discussion, we use $*$ as a place holder for any semantically valid comparison operator.

To define constraint conformance, we need to consider the different types of domains and all of the aspects that can be specified for a particular domain type. The first requirement for constraint conformance is that the domains are of the same type. More specifically, we allow conformance only for constraints of the same dimensions in two contract instances of a specific contract type.

To define conformance for aspects, we need to define the concept of *aspect signature*. An aspect signature is defined as the the tuple consisting of the aspect key, operator, and aspect the parameter if any. According to this we define the signature for the aspect **percentile** $P < V$ as the tuple $(\textbf{percentile}, <, P)$ and the signature for the aspect **frequency** $R > P\%$ as $(\textbf{frequency}, >, R)$. Aspect signatures of mean and variance are defined as $(\textbf{mean}, *)$ and $(\textbf{variance}, *)$ respectively. Let us assume that a constraint S conforms to another dimension constraint D of the same type. Then the rules in Figure 17 must hold.

The conforming constraint may introduce aspects that do not exist in the other

```
type ctName = contract {
  sd : decreasing set { e1, e2, e3, e4 } with order {e2<e1, e1<e3, e3<e4};
  ed : decreasing enum { a1, a2, a3 } with order {a1<a2, a2<a3};
  nd : decreasing numeric msec;
};

A = ctName contract {
    sd == {e2, e3};
    ed {
        frequency a1 > 10 %;
        frequency a3 < 10 %;
      };
    nd {
        percentile 90 < 18;
        percentile 100 < 23;
        mean <= 40;
        variance < 0.7;
}
};

B = ctName contract {
    sd == {e1};
    ed {
        frequency a1 > 20 %;
        frequency a3 < 10 %;
      };
    nd {
        percentile 10 < 5;
        percentile 50 < 10;
        percentile 90 < 18;
        percentile 100 <= 20;
        mean <= 40;
        variance < 0.6;
      };
};
```

Fig. 18. Conforming dimensions

constraints as long as it conforms to all the existing ones according to the above rules.

The example in Figure 18 shows a contract type and two contracts A and B of that same type. In this particular example, B conforms to A since each of the dimension constraints in B conform to the corresponding constraints in A.

The sd constraint in B conforms to sd in A since the element e1 in the set is a stronger the weakest element (e3) of sd in A. The dimension ed in B conforms since the aspect signature matches and since the lower bound of the constraint is increased. Finally, nd conforms because the distribution requires stronger values for each percentile that also exists in A and because the variance is lower. The mean is the same, which also gives rise to conformance.

Figure 19 shows another contract C of type ctName. The contract C does not conform to A. As a matter of fact, none of the constraints in C conform to the

```
C = ctName contract {
    sd == {e2, e4};
    ed {
        frequency a1 > 20 %;
        frequency a3 < 30 %;
      };
     nd {
        percentile 100 <= 23;
        mean <= 40;
        variance < 0.8;
      };
};
```

Fig. 19. Non-conforming dimensions

corresponding constraints in A. sd does not conform since the value contains an element that is weaker than any of the elements of the sd value in A. The ed value does not conform since we allow a higher frequency of the a3 value. Finally, for nd we set only an upper bound and remove the constraint in the $90th$ percentile, which results in a weaker constraint. The constraint over the mean aspect does conform, but we introduce a higher variance, which does not conform.

4. AN EXTENSION OF THE UNIFIED MODELING LANGUAGE

Profiles can be used to characterize QoS requirements and guarantees in many different situations. One example is run-time negotiation and monitoring of QoS agreements. In such situations we establish QoS agreements dynamically for limited time-periods, such as, in a session, or for a specified number of invocations. In other situations we define statically—as part of the design—the QoS that is required by various components in the system. Both situations require that a profile can be bound to a relation between a client and a server. In the dynamic case we bind to a deal as a result of a negotiation, while the static case requires that we bind as part of the design model. We consider the binding constructs quite context dependent. In this paper we will describe only a binding mechanism for object-oriented design.

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 [Booch et al. 1997] to support the definition of QoS properties. Later, we will use CORBA IDL [Object Management Group 1995] and our extension of UML [Booch et al. 1997] to describe an example design that includes QoS specifications.

In UML, *classes* are represented by rectangles. In addition, UML has a *type* concept that describes abstractions without providing an implementation. A type is drawn as classes with a type stereotype annotation (¡¡type¿¿) 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. Figure 20 shows a client using (dashed 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 extensions to UML allow QoS profiles to be associated with the *uses* and *implements* relationships between classes and interfaces. A reference to a profile is drawn as a rectangle with a dashed border within which the profile name is written. This profile box is then associated with a uses or implements relationship.

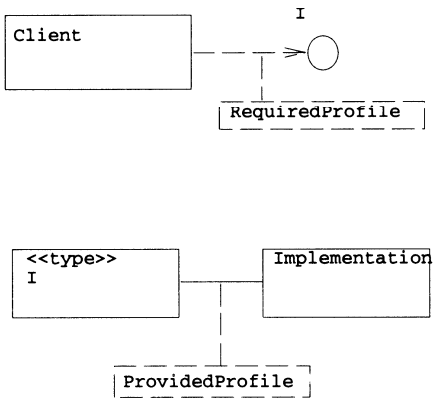

Fig. 20. UML extensions

In Figure 20, 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 QML.

Our UML extensions allow object-oriented design to be annotated with profile names that refer to separately defined QoS profiles. Notice that our UML extensions associate 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.

Having a separate graphical entity for QoS profiles allows us to clearly show when the same profile is referenced to from multiple places. For example, if a client requires the same profile that a service provides, we can graphically connect both the client and the service to the same profile box. In addition, we can show refinement relationships between profiles by introducing a graphical representation of these relationships.

5. 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 application level QoS requirements into QoS properties for application

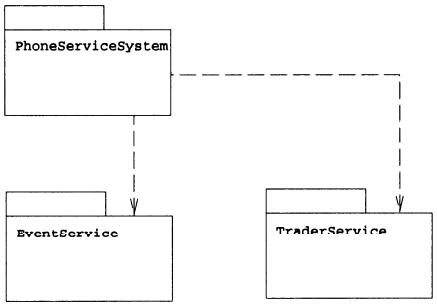

Fig. 21. High-level architecture

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 and events from the 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 in [Gray and Reuter 1993], we want the telephony service to be a ***highly-available*** system, which means it should have a total maximum downtime of five minutes per year. The availability measure for this downtime would 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 [Object Management Group 1995] Object Request Broker (ORB) to achieve scalability and reliability through distribution.

5.1 System Architecture

We call the service execution system module `PhoneServiceSystem`. As illustrated by Figure 21, it uses an `EventService` module and a `TraderService` module.

In Figure 22 we open up the `PhoneServiceSystem` module to see its main

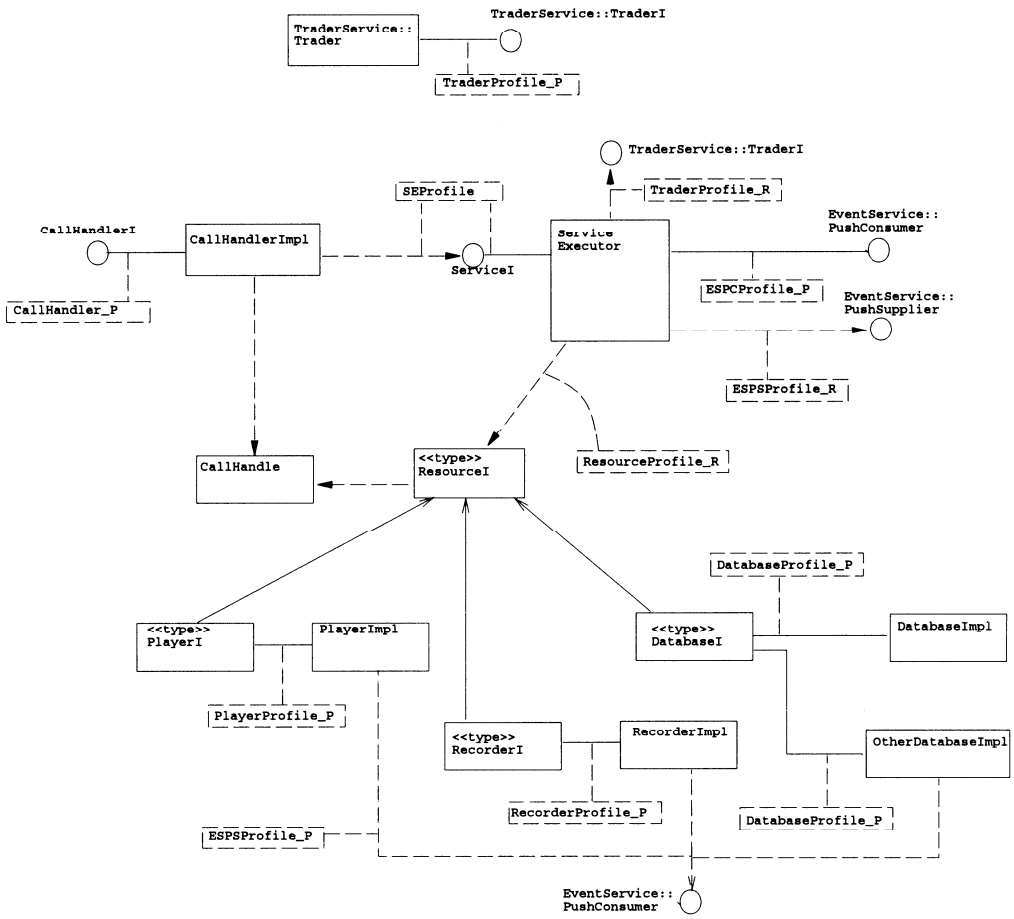

Fig. 22. Class diagram for PhoneServiceSystem

classes and interfaces. Classes are drawn as rectangles and interfaces as circles. Classes implement and use interfaces. The upper right part of Figure 22 shows that `ServiceExecutor` implements `ServiceI` and uses `TraderI`. In the diagram we have included references to QML profiles—such as `PlayerProfile_P` (lower left corner)— of which a subset will be described in section 5.2. 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 drawing some interrelationships to keep 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` (upper right corner).

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 and a `CallHandle` as arguments. The `CallHandle` 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` whenever an incoming call is received. The information stored 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 by contacting the trader and registering its offer when the resource is initially started. To reduce complexity of the diagram we omit showing that the resources use the trader.

`ServiceExecutor` uses the `PushSupplier` and implements the `PushConsumer` interface from 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 enables 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.

The diagram in Figure 22 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.

Other components, such as the `Trader`, are expected to pre-exist and therefore have previously specified QoS properties. With these components, one contract specifies the required properties and another contract specifies what is provided. Consequently we need to make sure the provided characteristics satisfy the required characteristics; this is referred to as *conformance* and is discussed in section 3.6.

We will now present simplified versions of three main interfaces in the design. The `ServiceI` interface described in Figure 23 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 contains channel identifiers and the other data necessary to execute the service.

Figure 24 defined the `TraderI` that is implemented by traders. The `TraderI` allows resources to offer and withdraw their services. Service executors can invoke the `find` or `findAll` operations on the `TraderI` to locate the 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 mal-

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

Fig. 23. The ServiceI interface

```
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 oi) raises (noMatch);
};
```

Fig. 24. The TraderI interface

functioning or depreciated resources. Observe that this is a much simplified trader for the purpose of this paper.

Finally, in Figure 25 we have the `PlayerI` that represents a simple player resource. Players allow the service execution 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`.

5.2 Reliability

We have already shown in Figure 22 how profiles are associated with *uses* and *implements* relationships between interfaces and classes. We will now discuss in more depth what the QoS profiles and contracts should be for this particular design. For the contracts we will use the dimensions proposed in Figure 9 of section 3.2. We will not present any development process by which the important profiles and their content can be identified.

To meet end-to-end reliability requirements, the underlying communications infrastructure, as well as the execution system, must meet reliability expectations. Making the simplifying assumption that the communications infrastructure is reliable, we focus on the reliability of the service execution system. In a real design we would take infrastructure reliability into account by comparing what a client requires with the composite reliability of the infrastructure and the server.

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

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

Fig. 25. The PlayerI interface

```
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;
  numOfFailure <= 2  failures/year;
  operationSemantics == atMostOnce;
};


CallHandlerProfile_P for CallHandlerI = profile {
  require CallServerReliability;
}
```

Fig. 26. Contract and binding for CallHandler

users, the `CallHandlerI` service must be highly available.

To provide a highly available telephone service, the `CallhandlerImpl` must have a very short recovery time and a long time between failures. Due to the shopping behavior of telephone service users, we must require both the repair time (`MTTR`) to not significantly exceed two minutes and the variance to be 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 2 failures per year on average. If the service fails, any executing and pending requests are discontinued and removed. This means we have a `at most once` operation semantics. The contract and profile of `CallHandlerI` as provided by `CallHandlerImpl` is described in Figure 26.

From Figure 22 we can see that the reliability of `CallHandlerI` depends directly on the reliability of service defined by `ServiceI`. The contract and profile used for `ServiceI` are described in Figure 27.

The `ServiceExecutor` component cannot 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 as simple as possible; therefore, we propagate high-availability requirements from `CallHandlerI` to the trader and the resources. This is certainly a major design decision that will affect the design and implementation of the other components of this system.

We expect the `ServiceExecutor` to have a short recovery time since it holds no information that needs to be recovered. When it fails, the service interactions it was

```
ServiceExecutorReliability = Reliability contract {
  MTTR < 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;
  numOfFailure <= 10 failures/year;
  operationSemantics == atMostOnce;
};

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

Fig. 27. Contract and binding for service

executing will be discontinued. We assume that users consider it more annoying if a session is interrupted by 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 three minutes on average with 80% of the calls lasting less than five minutes. With this in mind, we will require that the service executor provides high, continuous availability, which is reflected in the fact that a TTF of less than five minutes is unacceptable.

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 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 re-binding. 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.

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 be capable of executing when the system is up, which means that all resources must be available and must consequently satisfy the high-availability requirements.

The reliability contract for the `Trader` (defined in Figure 28) is based on a general contract (`HAServiceReliability`) for highly-available services. This contract is

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


TraderProfile_P for TraderI = profile {
  require HAServiceReliability refined by {
     MTTR {
            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 };
  };
};
```

Fig. 28. Contract and binding for the Trader

abstract in the sense that it states only 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, and consequently the number of failures can be allowed to be slightly higher.

In addition, we specify that offer identifiers and object references returned by the trader be valid even after a failure. This means that the offer identifier returned before a failure can be used to withdraw an offer, once 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 referenved have not failed.

The start-up time for a service execution is very important; the time between answerng a call and executing the service, must be short and must definitely not exceed 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 a dissatisfied customers and lost business opportunities.

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 throughput to constitute a bottleneck if this is done.

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. Performance requirements are specified in Figure 28 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 average TTFs to satisfy the high availability requirements. This does not mean that individual resources cannot fail, but it does mean that there must be sufficient redundancy to mask these failures.

In Figure 29 we define a general contract, called `ResourceReliability`, for `ResourceI`. The contract captures the requirements that resources need to be highly available. Each specific resource type will specify its individual QoS properties by refining this general contract.

The `PlayerReliability` contract in Figure 29 is defined as a refinement of `ResourceReliability`. It specifies that `PlayerImpl` will provide the high-availability specified in `ResourceReliability` by providing a long average `TTF`. We expect it to take two hours to repair a major problem with the player service. `PlayerImpl` provides extremely good continuous availability. The `TTF` characterization also indicates that the player very should rarely fail in a way that interrupts a service execution.

5.3 Discussion

The specification of reliability and performance contracts and the analysis of inter-component QoS dependencies have given us many important insights and guidance. For example, they have helped us realize that the `Trader` needs to support fast fail-over and use a reliable storage. We have also found that the reliability of resources is essential, and that, in this example system, resource services should be responsible for their own reliability.

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 allow 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, to estimate the frequency of failures due to programming errors, to consider communication link reliability, etc. We also need to ensure that the QoS contracts provided by components actually allow the clients to satisfy the requirements imposed on them. There are various modeling techniques available that are applicable to selected types of systems (see

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

PlayerReliability = ResourceReliability refined by {
  MTTR = 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;
};
```

Fig. 29. Contract and binding for resources

[Reibman and Veeraraghavan 1991] for an overview).

In our example case, high availability requirements for `CallHandler` result in strong demands on other services in the application. Another design alternative would be to demand that components such as the `ServiceExecutor` could handle failing 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, since 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 designers capture QoS requirements and associate them with interfaces at a detailed level. We also need to make first class citizens out of QoS requirements and offers from a design language point of view.

Finally, we believe the example shows that QML is suitable as a language for

specifying QoS constraints. In this example, we have focused on design time characterization, but the same QML concepts are suitable for QoS characterization at many different point in the system life-cycle.

6. SEMANTICS

Here we will more rigorously define selected areas of QML's semantics. The goal of these definitions is to verify the soundness and consistency of the language and to assist in the language implementation. Concerning language implementation, this work will primarily support the static semantic check in the compiler, but it will also support the algorithms for conformance checking.

We follow Tennent's [Tennent 1991] advice and attempt to provide a rigorous framework for understanding the semantics of QML in an informal setting:

> Mathematical rigor should not be confused with formality, which is certainly not sufficient for rigor. ... For most purposes it is actually preferable to use relatively informal descriptions, provided these are based on rigorous theoretical analysis, just as mathematicians normally present rigorous mathematics in an informal style.
> *R. D. Tennent*

Our semantic definition has two parts. We define a type system for QML. The inference rules of the type system specify when we can assign a given type to a given term. The second part of the definition is an interpretation over declarations that determine their soundness. The primary goal of this interpretation is to capture the soundness of refinement and conformance.

Notice that we cannot easily capture the rules for contract refinement and conformance as part of the type system. The reason is that the contract being refined and the result of the refinement are of the same type—we do not assign a different type to a contract defined by refinement. This is in contrast to object-oriented subtyping for which subtype soundness can be captured in a subtype relationship between different types. Similarly, conformance rules are checked for pairs of contracts of the same type. Thus, type rules are not sufficient to capture conformance semantics.

6.1 Notation

In describing the semantics, we use the following notation for sets and maps. $\mathrm{P}_{\omega}[X]$ is the set of finite subsets of X. The set of maps from X to Y is denoted $\mathrm{I}_{map}[X, Y]$, and $\mathrm{F}_{map}[X, Y]$ is the set of finite maps from X to Y. We use *Dom* to obtain the domain of a map.

We use "$[n \mapsto v]$" to describe a one-element map that maps n to v. We use $\bullet$ for map composition:

$$(A \bullet B)(i) = \begin{cases} B(i) & \text{if } i \in Dom(B) \\ A(i) & \text{otherwise.} \end{cases}$$

6.2 A Type System for QML

We describe a type system that assigns a unique type to a contract expression *conExp* (*conExp* $\in$ ConExp). Figure 30 gives the structure of contract expressions.

As in the syntactic definition of QML, x_c refers to a variable that holds contract values, x_p refers to a variable that holds profile values, and y refers to a variable that holds a contract type. The elements x_p, x_c, and y are all typical elements of the set Var. The term n refers to a name (an element in Name).

To simplify the type system, we ignore the concepts of units because their static semantics is relatively straightforward and because incorporating them would give rise to a significant increase in the number of type inference rules.

conExp	::=	y **contract** $\{c_1 ; \ldots ; c_k;\}$
	|	x_c **refined by** $\{c_1 ; \ldots ; c_k;\}$
c	::=	*n cSpec*
cSpec	::=	*op v*
	|	$\{aspect_1 ; \ldots ; aspect_n;\}$
v	::=	*n*
	|	$\{n_1 , \ldots , n_k\}$
	|	*number*
aspect	::=	**percentile** *percentNum op dimValue*
	|	**mean** *op v*
	|	**variance** *op v*
	|	**frequency** *freqRange op percentNum* %
freqRange	::=	*dimValue*
	|	*lRangeLimit dimValue , dimValue rRangeLimit*
lRangeLimit	::=	(| [
rRangeLimit	::=	) |]
op	::=	== | >= | <= | < | >
percentNum	::=	0 | 1 | ... | 99 | 100
number	::=	...

Fig. 30. Structure of expressions and values

We consider two kinds of types: dimension and contract. A dimension describes a dimension's domain, including the values in the domain and the ordering of the domain. A contract type gives the common structure of a set of contracts that are instances of the type. Contract types are elements of ConType (*conType* $\in$ ConType). The set Type is the set of all types (contract and dimension). We use the names t and *type* to refer to elements in Type. The structure of types is described in Figure 31.

The inference rules state when we can consider a given term to be of a given type. The terms under consideration are contract expressions (including contract values) and constraints. Types are assigned relative to a *type context*, called Γ in the inference rules. The notion of a type context reflects the fact that terms are typed based on previous type assignments. The type context can be thought of as the "state" of type analysis at any given point.

Type contexts have the following structure:

$$\Gamma ::= x_c : t \mid y = t \tag{1}$$

The type context of the form "$x_c : t$" reflects a previous type assignment of the type t to the variable x_c. In QML, the variable x_c would be the name of a contract

```
type      ::= dimType | conType
conType   ::= contract {n1 : dimType1 ; ... ; nk : dimTypek}
dimType   ::= enum {n1 , ... , nk}
            | relSem enum {n1 , ... , nk} with order
            | set {n1 , ... , nk}
            | relSem set {n1 , ... , nk}
            | relSem set {n1 , ... , nk} with order
            | relSem numeric
order     ::= order {ni < nj , ... , nk < nm}
relSem    ::= decreasing | increasing
```

Fig. 31. Structure of types

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

Fig. 32. Explicit typing of expression

and t would be the type of this contract. A context of the form "$y = t$" reflects a type declaration of the form "**type** $y = t$" where y is declared as a type name of the type t.

We require that two contract type declarations do not use the same type name. In other words, we require that a typing context has only one typing assignment of the form "$y = t$" for any name y. To keep the descriptions simple, the typing rules do not currently express this property of type contexts. They could, however, be extended to enforce this property.

We do allow literals of different types to be identical. For example, consider two enums, both with the value v. We allow this because the value v is used only in the context of a specific dimension for which the type is unambiguously defined; thus, the type of v can be derived.

The most basic inference rule states that the type of variables is determined by the type context:

$$\Gamma, x_c : t \vdash x_c : t \tag{2}$$

Informally, the rule states that if the type context contains a binding "$x_c : t$" we can conclude that the variable x_c is bound to the type t.

The goal of type inference for QML contracts is to determine the type of a contract and to determine its correctness with respect to its declared contract type. The example in Figure 32 shows how a contract value can be defined within a profile and how to associate an explicit type. In the figure, we explicitly associate the type `Reliability` with the contract value "**contract** {`rebindPolicy == noRebind`}."

The typing rules for contract expressions are given below. The first rule (rule

(3)) determines when a contract value with k dimensions can be considered to be of a contract type with k dimensions. The rule states that the dimension names must be the same in the contract value and the contract type. Moreover, the constraint specification for a dimension with name n must be of type t, where t is the dimension type for n in the contract type.

The second rule (rule (4)) captures the property that a contract value can omit specification along certain dimensions. That is, a contract value may contain fewer dimensions than its contract type as long as the dimension constraints have the correct type.

The third rule (rule (5)) defines how we can determine type correctness for an explicitly typed contract expression. We can infer that "y **contract**$\{\ldots\}$" is of type t, if y has been previously bound to t and "**contract**$\{\ldots\}$ is of type t. The typing rules assign unique types to contract expressions. This property is primarily due to the explicit typing of contract values in QML. Rule (5) connects the explicit typing of contract values to a requirement on type declarations reflected in the type context.

The last rule (rule (6)) specifies the circumstances under which we can consider a refined contract to be of a given type. Both the delta and the contract being refined must be of the same type t.

$$\frac{\Gamma \vdash n_1\ cs_1 : t_1 \quad \ldots \quad \Gamma \vdash n_k\ cs_k : t_k}{\Gamma \vdash \textbf{contract}\ \{n_1\ cs_1\ ;\ldots;\ n_k\ cs_k\} : \textbf{contract}\ \{n_1 : t_1\ ;\ldots;\ n_k : t_k\}} \tag{3}$$

$$\frac{\Gamma \vdash \textbf{contract}\ \{c_1\ ;\ldots;\ c_k\} : \textbf{contract}\ \{n_1 : t_1\ ;\ldots;\ n_k : t_k\}}{\Gamma \vdash \textbf{contract}\ \{c_1\ ;\ldots;\ c_k\} : \textbf{contract}\ \{n_1 : t_1\ ;\ldots;\ n_k : t_k\ ;\ n_{k+1} : t_{k+1}\ ;\ldots;\ n_m : t_m\}} \tag{4}$$

$$\frac{\Gamma \vdash \textbf{contract}\ \{c_1\ ;\ldots;\ c_k\} : t}{\Gamma, y = t \vdash (y\ \textbf{contract}\ \{c_1\ ;\ldots;\ c_k\}) : t} \tag{5}$$

$$\frac{\Gamma \vdash x_c : t \qquad \Gamma \vdash \textbf{contract}\ \{c_1\ ;\ldots;\ c_k\} : t}{\Gamma \vdash (x_c\ \textbf{refined by}\ \{c_1\ ;\ldots;\ c_k\}) : t} \tag{6}$$

The above typing rules ignore the following aspects of contracts and contract types:

—The order in which dimensions appear in contracts or contract types is not significant. We do not capture this fact in the above rules. In fact, we assume a specific dimension order in rules (3) and (4). It would be more precise, but also more complicated, if these rules handled arbitrary permutations of dimensions.

—We do not capture the constraint that a contract or contract type cannot have two dimensions with the same name. We could possibly extend the typing rules to assign types only to contracts for which this constraint is satisfied. To express the constraint for contract types, we need to specify the semantics for the declaration of contract types, which is not part of the type system.

To use the rules (3), (4), (5), and (6), we need to determine type correctness for individual dimension constraints. We introduce rules to determine type correctness

for simple constraints such as "*delay* < 10," as well as for more complicated constraints involving multiple aspects such as "*delay* $\{$**mean** < 12; **variance** $< 0.33\}$." Type correctness for constraints is determined by primitive types.

The type inference rules for simple constraints over enumeration domains are described below. The rules determine type correctness for an expression involving a name literal n_i. As an example, assume that we are checking correctness of the expression $n == n_i$, where n is a dimension name. The first rule (rule (7)) states that if the name n_i is in the set "$\{n_1, \ldots, n_m\}$" defined in the definition "**enum** $\{\ldots\}$" of n, we can infer that the expression "$n == n_i$" is of type "**enum** $\{n_1, \ldots, n_m\}$."

The following two rules capture the semantic property that the operators allowed in constraint expressions depend on whether the dimension is increasing or decreasing. These rules may seem ambiguous with respect to the $==$ operator, but are in fact not. We select rules based on the dimension type, and each rule has a distinct dimension type.

$$\frac{n_i \in \{n_1, \ldots, n_m\}}{\Gamma \vdash n == n_i : \textbf{enum } \{n_1, \ldots, n_m\}} \tag{7}$$

$$\frac{n_i \in \{n_1, \ldots, n_m\} \quad op \in \{<, <=, ==\}}{\Gamma \vdash n \ op \ n_i : \textbf{decreasing enum } \{n_1, \ldots, n_m\} \textbf{ with } order} \tag{8}$$

$$\frac{n_i \in \{n_1, \ldots, n_m\} \quad op \in \{>, >=, ==\}}{\Gamma \vdash n \ op \ n_i : \textbf{increasing enum } \{n_1, \ldots, n_m\} \textbf{ with } order} \tag{9}$$

According to these rules, we can use inequality constraints only for ordered enumeration domains. Furthermore, the name (n_i) used as part of a dimension constraint must belong to the domain specified for the dimension.

The following rules determine type correctness for constraints over set domains. The first rule (rule (10)) states that if we use a set of names in a constraint, those names must belong to the constraint's domain. The second rule (rule (11)) states that we can use only the operators "$<, <=, ==$" for constraints over a decreasing domain. Moreover, the second rule also ensures that the names in a set constraint must belong to the constraint's domain.

$$\Gamma \vdash n == \{n_1, \ldots, n_k\} : \textbf{set } \{n_1, \ldots, n_k, n_{k+1}, \ldots, n_m\} \tag{10}$$

$$\frac{op \in \{<, <=, ==\}}{\Gamma \vdash n \ op \ \{n_1, \ldots, n_k\} : \textbf{decreasing set } \{n_1, \ldots, n_k, n_{k+1}, \ldots, n_m\}} \tag{11}$$

$$\frac{op \in \{>, >=, ==\}}{\Gamma \vdash n \ op \ \{n_1, \ldots, n_k\} : \textbf{increasing set } \{n_1, \ldots, n_k, n_{k+1}, \ldots, n_m\}} \tag{12}$$

$$\frac{op \in \{<, <=, ==\}}{\Gamma \vdash n \ op \ \{n_1, \ldots, n_k\} : \textbf{decreasing set } \{n_1, \ldots, n_k, n_{k+1}, \ldots, n_m\} \textbf{ with } order} \tag{13}$$

$$\frac{op \in \{>, >=, ==\}}{\Gamma \vdash n\ op\ \{n_1, \ldots, n_k\} : \\ \quad \textbf{increasing set}\ \{n_1, \ldots, n_k, n_{k+1}, \ldots, n_m\}\ \textbf{with}\ order} \tag{14}$$

As for enumeration domains, we can use only inequality constraints for set domains that are ordered. For sets we have three cases: no declarations; increasing or decreasing set; or ordered increasing/decreasing set.

We consider two types of numeric domains: increasing, for which bigger values are stronger, and decreasing, for which smaller values are stronger. Numeric domains are always totally ordered. As for enumeration and set constraints, we allow only constraints with respect to the weakest allowed value. That is, we can describe only constraints that can be strengthened: we allow only lower bounds for increasing and upper bounds for decreasing.

The typing rules for numeric constraints are listed below. The purpose of the rules is to capture the type correctness of numeric constraints, such as "$n < 4$," relative to a primitive type, such as "**increasing numeric**." The first rule (rule (15)) states that if we have <, <=, or ==, and the dimension is decreasing, we can infer that the dimension type of the constraint must be decreasing numeric.

$$\frac{op \in \{<, <=, ==\}}{\Gamma \vdash n\ op\ number : \textbf{decreasing numeric}} \tag{15}$$

$$\frac{op \in \{>, >=, ==\}}{\Gamma \vdash n\ op\ number : \textbf{increasing numeric}} \tag{16}$$

Next we introduce the rules to determine type correctness for aspects. The first rule (rule (17)) states that an aspect clause is of type t if all the constituent aspects of the clause are of type t. The subsequent rules define the type constraints for individual aspects.

$$\frac{\Gamma \vdash aspect_1 : t \quad \ldots \quad \Gamma \vdash aspect_k : t}{\Gamma \vdash n\{aspect_1; \ldots; aspect_k\} : t} \tag{17}$$

$$\frac{\exists\, n \in \mathsf{Name}, op_1 \in \{<, >, <=, >=, ==\} : \Gamma \vdash (n\ op_1\ v) : t}{\Gamma \vdash \textbf{frequency}\ v\ op\ percentNum\ \% : t} \tag{18}$$

$$\frac{\begin{array}{c}\exists\, n_1, n_2 \in \mathsf{Name}, op_1, op_2 \in \{<, >, <=, >=\} : \\ \Gamma \vdash (n_1\ op_1\ v_1) : t \qquad \Gamma \vdash (n_2\ op_1\ v_2) : t\end{array}}{\Gamma \vdash \textbf{frequency}\ lRangeLimit\ v_1\ ,\ v_2\ rRangeLimit\ op\ percentNum\ \% : t} \tag{19}$$

$$\Gamma \vdash \textbf{percentile}\ percentNum\ op\ number : numRel\ \textbf{numeric} \tag{20}$$

$$\frac{op \in \{<, <=, ==\}}{\Gamma \vdash \textbf{variance}\ op\ number : numRel\ \textbf{numeric}} \tag{21}$$

$$\frac{op \in \{>, >=, ==\}}{\Gamma \vdash \textbf{mean}\ op\ number : \textbf{increasing numeric}} \tag{22}$$

$$\frac{op \in \{<, <=, ==\}}{\Gamma \vdash \textbf{mean } op \textit{ number} : \textbf{decreasing numeric}} \tag{23}$$

The second and third rule (rule (18) and (19)) deal with type correctness for frequencies. The second rule deals with frequencies of single values. The typing constraint is that the value v must belong to the domain of the aspect's dimension. The type t in the rule is a dimension type, that is, a type for constraints. Thus, we need to construct a constraint involving v that is of type t whenever v is in the dimension's domain. The constraint "n op_1 v," for arbitrary dimension name n and operator op_1, is of type t whenever v is in the domain of n.

The third rule deals with frequencies for intervals of values. The typing constraint is that the values v_1 and v_2 must both belong to the dimension's domain. Furthermore, the domain must be ordered. As for rule (18), we construct constraints that are of the proper type whenever the frequency aspect is of the proper type. The constraints are similar to the constraint constructed in rule (18), except that the operators must be inequality operators. The requirement of inequality operators ensures that the domain will be ordered—if the domain is not ordered, the constraints will not be type correct because we do not allow inequality operators over unordered domains.

6.3 Semantics of Contract Declaration

To describe the semantics of contract refinement and conformance, we construct a (denotational) mapping, called $\mathsf{contract}_{exp}$, from contract expressions to a domain called ContractRep. Elements in ContractRep represent contracts. We describe refinement as an operator on elements in ContractRep, and we describe conformance as a relation on elements in ContractRep.

Elements in ContractRep are finite maps from dimension names to either constraints or aspects:

$$C \in \mathsf{ContractRep} = \mathrm{F}_{map}[\mathsf{Name}, \mathsf{ConstraintSpec} \cup \mathsf{AspectRep} \cup \{\bot\}] \tag{24}$$

Notice that (24) both defines a domain (ContractRep = ...) and specifies a notation for a typical element in that domain ($C \in \mathit{ContractRep}$).

A map in ContractRep can map a dimension name to the element $\bot$. The element $\bot$ represents an illegally defined dimension. Illegal dimensions result from illegal use of refinement. We say that a contract is illegal if it contains at least one illegal dimension.

We represent constraints as elements in the domain ConstraintSpec:

$$cs \in \mathsf{ConstraintSpec} = \mathsf{Operator} \times \mathsf{DimValue} \tag{25}$$

$$op \in \mathsf{Operator} = \{==, >=, <=, <, >\} \tag{26}$$

An element cs in ConstraintSpec is a pair consisting of an operator and an element in DimValue. For example, we could represent the constraint "$\mathit{delay} < 10$" by the pair "$(<, 10)$."

The set DimValue contains the values over which we describe constraints. The DimValue set has three subsets: one that contains names (Name), one that contains sets of names (P_ω[Name]), and one that contains numeric values (Number).

$$v \in \mathsf{DimValue} = \mathsf{Name} \cup \mathrm{P}_{\omega}[\mathsf{Name}] \cup \mathsf{Number} \tag{27}$$

We represent aspects as elements of the domain AspectRep:

$$ar \in \mathsf{AspectRep} = \mathrm{F}_{map}[\mathsf{AspectName}, \mathsf{ConstraintSpec} \cup \{\bot\}] \tag{28}$$

$$\begin{aligned}\mathsf{AspectName} = {} & \{\mathsf{variance}, \mathsf{mean}\} \cup \\ & \{\mathsf{percentile}\} \times \mathsf{PercentNum} \times \mathsf{Bound} \cup \\ & \{\mathsf{frequency}\} \times \mathsf{FreqRange} \times \mathsf{Bound}\end{aligned} \tag{29}$$

$$\mathsf{Bound} = \{\mathsf{lower}, \mathsf{upper}\} \tag{30}$$

An element in AspectRep is a map from aspect names to constraints. An aspect name is a tuple that identifies a particular aspect of a dimension. For example, an aspect could be the 90th percentile. The constraints over percentiles can specify both upper and lower bounds (using $<$ and $>$). Thus, we consider the lower and the upper bounds to be two different aspects. For mean and variance, we can specify only an upper or a lower bound, but not both. Our notion of aspect name corresponds to the notion of aspect signature in Section 3.6.1.

The domain FreqRange is the set of syntactic terms that describe frequency ranges. Such terms are either single values (elements in DimValue, or they are intervals of values, such as $[3, 7]$.

Having sketched the domains that represent contracts, we need to define a function that maps (syntactic) contract expressions to elements in ContractRep. We can then define contract refinement and conformance as operations and relations on ContractRep. The function $\mathsf{contract}_{exp}$ maps a contract expression to its representation in ContractRep.

$$\mathsf{contract}_{exp} : (\mathsf{ConExp} \times \mathsf{Env_c}) \rightarrow \mathsf{ContractRep} \tag{31}$$

$$\mathsf{contract}_{dim} : (\mathsf{Dimension} \times \mathsf{Env_c}) \rightarrow \mathsf{ContractRep} \tag{32}$$

$$\mathsf{contract}_{asp} : (\mathsf{Aspect} \times \mathsf{Env_c}) \rightarrow \mathsf{ContractRep} \tag{33}$$

The function $\mathsf{contract}_{exp}$ maps contract expressions to elements in ContractRep based on the structure of contract expressions. Thus, we need functions that map sub-structures to elements in ContractRep. The function $\mathsf{contract}_{dim}$ maps dimensions or constraints to elements in ContractRep, whereas $\mathsf{contract}_{asp}$ maps aspects to elements in ContractRep.

The set Decl is the set of syntactic terms that capture QML declarations. We consider only declaration of contracts—the issues concerned with declaration of contract types are covered by the type system. A declaration is then of the form "$x_c = conExp$".

The set $\mathsf{Env_c}$ is the set of environments, where an environment is a map from variables to elements in ContractRep. The notion of environment is necessary to capture the impact of previous declarations on the current declaration.

$$E_c \in \mathsf{Env_c} = \mathrm{F}_{map}[\mathsf{Var}, \mathsf{ContractRep}] \tag{34}$$

$$\mathsf{check} : (\mathsf{Decl} \times \mathsf{Env_c}) \rightarrow \mathsf{Bool} \tag{35}$$

$$\mathsf{legal_{con}} : \mathsf{ContractRep} \rightarrow \mathsf{Bool} \tag{36}$$

The check function checks a list of declarations relative to an environment. It returns true, if the all declarations are correct, and false otherwise.

$$\begin{aligned} &\mathsf{check}((x_c = conExp; decl), E_c) = \\ &\quad \mathsf{legal_{con}}(\mathsf{contract}_{exp}(conExp, E_c)) \ \wedge \\ &\qquad \mathsf{check}(decl, E_c \bullet [x_c \mapsto \mathsf{contract}_{exp}(conExp, E_c)]) \end{aligned} \tag{37}$$

The function $\mathsf{legal_{con}}$ determines legality for an individual element in ContractRep.

$$\mathsf{legal_{con}}(C) = \forall i \in Dom(C) : C(i) \neq \bot \tag{38}$$

The strategy is to map the contract expression of an individual declaration to its representation in the ContractRep set, and then determine the legality of the declaration by inspecting this representation through $\mathsf{legal_{con}}$. The rules below assume that declarations are type correct, and they primarily detect errors with respect to refinement.

The function $\mathsf{contract}_{exp}$ is defined below in terms of pattern matching on the structure of contract expressions. The first rule matches contract definitions, where y denotes the name of the associated contract type. The rule essentially states that to check a contract we need to check every dimension using $\mathsf{contract}_{dim}$. Refinement is captured by rule (41).

$$\begin{aligned} &\mathsf{contract}_{exp}(y \ \mathbf{contract} \ \{c_1 ; \ldots ; c_k\}, E_c) = \\ &\quad \mathsf{contract}_{dim}(c_1, E_c) \bullet \ldots \bullet \mathsf{contract}_{dim}(c_k, E_c) \end{aligned} \tag{39}$$

$$\mathsf{contract}_{exp}(x_c, E_c) = E_c(x_c) \tag{40}$$

$$\begin{aligned} &\mathsf{contract}_{exp}(x_c \ \mathbf{refined\ by} \ \{c_1 ; \ldots ; c_k\}, E_c) = \\ &\quad E_c(x_c) \lhd_t (\mathsf{contract}_{dim}(c_1, E_c) \bullet \ldots \bullet \mathsf{contract}_{dim}(c_k, E_c)) \quad \text{where } x_c : t \end{aligned} \tag{41}$$

$$\mathsf{contract}_{dim}(n_1 \ op \ n_2, E_c) = [n_1 \mapsto (op, n_2)] \tag{42}$$

$$\mathsf{contract}_{dim}(n \ op \ \{n_1, \ldots, n_k\}, E_c) = [n \mapsto (op, \{n_1, \ldots, n_k\})] \tag{43}$$

$$\mathsf{contract}_{dim}(n \ op \ number, E_c) = [n \mapsto (op, number)] \tag{44}$$

$$\begin{aligned} &\mathsf{contract}_{dim}(n \ \{a_1; \ldots ; a_k\}, E_c) = \\ &\quad [n \mapsto (\mathsf{contract}_{asp}(a_1, E_c) \bullet \ldots \bullet \mathsf{contract}_{asp}(a_k, E_c))] \end{aligned} \tag{45}$$

The operator $\lhd_t$ represents contract refinement relative to a contract type t. We need to consider contract types because type information, such as the ordering of elements in enums and sets, influences the semantics of contract refinement. If we have two elements C_1 and C_2 in ContractRep, the expression "$C_1 \lhd_t C_2$" represents C_1 refined by C_2. We define $\lhd_t$ below.

The $\mathsf{contract}_{asp}$ rules below define the mapping from aspects to elements in ContractRep. The domain of an element for frequency and percentile in ContractRep contains either upper or lower in order to allow both types of constraints to be defined as part of the same contract. The value in an equality constraint (==) defines both an upper and lower a bound.

$$\mathsf{contract}_{asp}(\textbf{frequency } fr \ == \ pNum \ \%, E_c) = \\ [(\mathsf{frequency}, v, \mathsf{lower}) \mapsto (==, pNum)] \bullet \\ [(\mathsf{frequency}, fr, \mathsf{upper}) \mapsto (==, pNum)] \tag{46}$$

$$\mathsf{contract}_{asp}(\textbf{frequency } fr \ op \ pNum \ \%, E_c) = \\ [(\mathsf{frequency}, fr, \mathsf{lower}) \mapsto (op, pNum)] \quad \text{if } op \in \{>, >=\} \tag{47}$$

$$\mathsf{contract}_{asp}(\textbf{frequency } fr \ op \ pNum \ \%, E_c) = \\ [(\mathsf{frequency}, fr, \mathsf{upper}) \mapsto (op, pNum)] \quad \text{if } op \in \{<, <=\} \tag{48}$$

$$\mathsf{contract}_{asp}(\textbf{percentile } pNum \ == \ number, E_c) = \\ [(\mathsf{percentile}, pNum, \mathsf{lower}) \mapsto (==, number)] \bullet \\ [(\mathsf{percentile}, pNum, \mathsf{upper}) \mapsto (==, number)] \tag{49}$$

$$\mathsf{contract}_{asp}(\textbf{percentile } pNum \ op \ number, E_c) = \\ [(\mathsf{percentile}, pNum, \mathsf{lower}) \mapsto (op, number)] \quad \text{if } op \in \{>, >=\} \tag{50}$$

$$\mathsf{contract}_{asp}(\textbf{percentile } pNum \ op \ number, E_c) = \\ [(\mathsf{percentile}, pNum, \mathsf{upper}) \mapsto (op, number)] \quad \text{if } op \in \{<, <=\} \tag{51}$$

$$\mathsf{contract}_{asp}(\textbf{variance } op \ number, E_c) = [(\mathsf{variance}) \mapsto (op, number)] \tag{52}$$

$$\mathsf{contract}_{asp}(n, \textbf{mean } op \ number, E_c) = [(\mathsf{mean}) \mapsto (op, number)] \tag{53}$$

Having defined the structural mapping of contract expressions to elements in ContractRep, we can now capture the rules for refinement. We model refinement as the operator $\lhd_t$ defined on ContractRep; it is defined as follows:

$$\text{for } C_1, C_2 \in \mathsf{ContractRep},\ t \in \mathsf{ConType},\ \text{and } n \in Dom(C_1) \cup Dom(C_2) : \\ (C_1 \lhd_t C_2)(n) = \begin{cases} C_1(n) & \text{if } n \in (Dom(C_1) \setminus Dom(C_2)) \\ C_2(n) & \text{if } n \in (Dom(C_2) \setminus Dom(C_1)) \\ C_2(n) & \text{if } C_2(n) \prec_{(n,t)} C_1(n) \\ \bot & \text{otherwise.} \end{cases} \tag{54}$$

Here, n is the name of a dimension. If there is a dimension value for n in only one of the involved contracts, then that value is the result of refinement along the dimension n. If both contracts specify a value for n, then the delta contract (C_2 in the above rules) must specify a stronger constraint than the contract being refined (C_1 in the above rules). If the constraint in the delta is indeed stronger, it will be the result of refinement. Refinement amounts to replacing weaker constraints

with stronger constraints. The *stronger than* relation is normally referred to as *conformance* and is defined in the following section.

6.4 Constraint Conformance

The most semantically intricate aspects of QML are refinement and conformance of profiles and contracts. To define these we need a precise understanding of when one constraint satisfies another.

Constraint conformance defines when one dimension constraint conforms to another dimension constraint. A dimension constraint conforms to another dimension constraint only if it is stronger than, or equally strong as, the other constraint. Here *stronger than* is relative to the type declaration and ordering given for the dimension. As an example, consider a dimension defined as "*delay* : **decreasing numeric**." Then the constraint "*delay* $\{mean < 10\}$" also satisfies the weaker constraint "*delay* $\{mean < 20\}$."

First, we define conformance for aspects. Remember that we represent an aspect clause as a map from aspect names (AspectName)to operator-value pairs. The rule (55) states that an aspect clause A_1 conforms to another aspect clause A_2, if the domain of A_2 is a subset of the domain of A_1 and if each individual aspect in A_1 conforms to the corresponding aspect of A_2. Thus, a conforming aspect clause can add new aspects and strengthen only those aspects that exist in the aspect clause to which it is conforming.

The statement $A_1 \prec_{(n,t)} A_2$ denotes conformance of A_1 to A_2 with respect to dimension n of contract type t.

$$\frac{\begin{array}{l}\text{for } A_1, A_2 \in \mathsf{AspectRep}: \\ Dom(A_2) \subseteq Dom(A_1) \\ \forall\, a \in Dom(A_2) : A_1(a) \prec_{(n,t)} A_2(a)\end{array}}{A_1 \prec_{(n,t)} A_2} \tag{55}$$

We define aspect conformance in terms of conformance for simple constraints represented by operator-value pairs. We now define conformance for simple constraints. For simple constraints that have identical values, the following rules apply:

for $v \in \mathsf{DimValue}$:

$$(op, v) \prec_{(n,t)} (op, v) \tag{56}$$

$$(--, v) \prec_{(n,t)} (>-, v) \tag{57}$$

$$(>, v) \prec_{(n,t)} (>=, v) \tag{58}$$

$$(==, v) \prec_{(n,t)} (<=, v) \tag{59}$$

$$(<, v) \prec_{(n,t)} (<=, v) \tag{60}$$

The first rule (rule (56)) states that a constraint conforms to another identical constraint. The rules (57) to (60) define conformance for constraints with identical values but different operators. If the values are not identical, we have to describe conformance in terms of the domain ordering. We specify those rules for enumeration, set, and numeric domains below.

For enumeration domains, conformance for operators other than equality is possible only for ordered enums. In addition, one constraint is stronger than another only if the elements involved in the constraints are ordered and if the value in the conforming constraint is stronger than the value in the other constraint. Notice that we have different rules depending on the operator. We need to determine only the direction of op_2 since the type correctness requirement ensures op_1 has the same direction.

$$\text{for } n_1, n_2 \in \mathsf{Name}:$$

$$\frac{(n_1, n_2) \in \mathsf{order}(n, t)}{(op_1, n_1) \prec_{(n,t)} (op_2, n_2)} \quad op_2 \in \{<, <=\} \tag{61}$$

$$\frac{(n_2, n_1) \in \mathsf{order}(n, t)}{(op_1, n_1) \prec_{(n,t)} (op_2, n_2)} \quad op_2 \in \{>, >=\} \tag{62}$$

The following functions extract information about orders and set types from contract types along a given dimension. We represent orders as relations on Name. The function order constructs a canonical representation of the order specified for a dimension n in a contract type t. The function then returns the transitive closure of this canonical representation (we use "+" to denote a transitive closure).

$$\mathsf{Order} = \mathrm{P}_\omega[\mathsf{Name} \times \mathsf{Name}] \tag{63}$$

$$\mathsf{order} : (\mathsf{Name} \times \mathsf{ConType}) \to \mathsf{Order} \tag{64}$$

$$\mathsf{order}(n, t) =$$

$$\begin{cases} (\{(n_i, n_j)\} \cup \ldots \cup \{(n_k, n_m)\})^+ & \text{if } \mathbf{order}\{n_i < n_j, \ldots, n_k < n_m\} \\ & \text{is the order specified for } n \text{ in } t \\ \emptyset & \text{otherwise.} \end{cases} \tag{65}$$

For set values, the rules for conformance are different for decreasing sets and increasing sets:

$$\text{for } A, B \in \mathrm{P}_\omega[\mathsf{Name}]:$$

$$\frac{\forall a \in (A \setminus B) : \exists b \in (B \setminus A) : (a, b) \in \mathsf{order}(n, t)}{(op_1, A) \prec_{(n,t)} (op_2, B)} \quad op_2 \in \{<, <=\} \tag{66}$$

$$\frac{\forall b \in (B \setminus A) : \exists a \in (A \setminus B) : (a, b) \in \mathsf{order}(n, t)}{(op_1, A) \prec_{(n,t)} (op_2, B)} \quad op_2 \in \{>, >=\} \tag{67}$$

For two sets A and B of a decreasing dimension, A conforms to B if the elements in $A \setminus B$ are stronger than at least one element in $B \setminus A$. Similarly, for increasing sets, all elements in $B \setminus A$ must be stronger than at least one element in $A \setminus B$.

For numeric constraints, we define conformance in terms of the canonical ordering of the real numbers.

```
profileDecl   ::= x_p for intName = profileExp
profileExp    ::= profile
                | x_p refined by {req_1 ; ... ; req_n ;}
profile       ::= profile {req_1 ; ... ; req_n ;}
req           ::= require conList
                | from entityList require conList
contractList  ::= conExp_1 , ... , conExp_n
entityList    ::= entity_1 , ... , entity_n
entity        ::= opName
                | attrName
                | opName.parName
                | resultof opName
```

Fig. 33. Abstract syntax for profiles

$$\frac{\mathsf{number}(v_1) < \mathsf{number}(v_2)}{(op_1, v_1) \prec_{(n,t)} (op_2, v_2)} \quad op_2 \in \{<, <=\} \tag{68}$$

$$\frac{\mathsf{number}(v_1) > \mathsf{number}(v_2)}{(op_1, v_1) \prec_{(n,t)} (op_2, v_2)} \quad op_2 \in \{>, >=\} \tag{69}$$

The function number returns the mathematical value that corresponds to a syntactic representation of a number.

6.5 Semantics of Profile Declaration

We give a semantics for profiles and their declarations. For the reader's convenience, we repeat the abstract syntax for profiles and their declaration in Figure 33.

The emphasis of the semantics is to capture the concepts of a default contract within a profile, profile refinement, and profile conformance. The semantics for profiles assume that the syntactic terms satisfy the following static semantic properties:

—A profile can have at most one default contract of a given contract type.

—A profile can specify at most one individual contract of a given contract type for a specific entity.

We could possibly capture these constraints as part of the way we build up composite elements in ProfileRep from primitive elements returned by $\mathsf{defContracts}_{req}$ and $\mathsf{defContracts}_{req}$. One could, for example, define a modified map composition operator that does not allow the domains of maps in ProfileRep to overlap. We ignore these issues because we want to focus on the semantics of profile refinement and conformance.

As for contracts, we define a semantic domain in which profiles may be represented mathematically, and we define a (denotational) mapping from syntactic terms into that domain. The domain for profile representation is called ProfileRep.

$$P \in \mathsf{ProfileRep} = \mathrm{F}_{map}[\mathsf{Entity} \times \mathsf{Type}, \mathsf{ContractRep}] \tag{70}$$

In ProfileRep, a profile is represented as a map. The map for a particular profile p takes an entity e and a contract type t, and returns the contract of type t that applies to e within p. Remember that an *entity* is a common term for interface constituents such as operations, attributes, and parameters. Although strictly speaking elements of ProfileRep *represent* profiles, we sometimes refer to them simply as *profiles*.

The reason for mapping profile syntactic terms into elements in ProfileRep is that we can more conveniently describe the semantics of conformance as a relation on elements in ProfileRep. However, we do not describe refinement as an operation on elements in ProfileRep. To capture refinement, we need to distinguish, in the semantic domain, between default contracts and individual contracts within a profile.

We specify an individual contract in the following way: "**from** e **require** c." This requirements clause specifies a contract for the individual entity e. A default contract is specified as "**require** c." With this clause, c applies to all entities within the interface for which the profile is defined.

To capture the distinction between individual and default contracts, we define a semantic domain, called ProfContracts, in which profiles are represented a pais of maps. The first map in a pair represents the individual contracts and the second map in a pair represents the default contracts.

$$\mathsf{ProfContracts} = \mathsf{IndContracts} \times \mathsf{DefContracts} \tag{71}$$

$$IP \in \mathsf{IndContracts} = \mathsf{ProfileRep} \tag{72}$$

$$DP \in \mathsf{DefContracts} = \mathrm{F}_{map}[\mathsf{Type}, \mathsf{ContractRep}] \tag{73}$$

The map for default contracts only takes a contract type. The map for individual contracts takes both an entity and a contract type.

Our approach in defining the semantics of profiles is to map profile expressions into the domain ProfContracts. The function called $\mathsf{profContracts}_{exp}$ captures this mapping. We describe refinement as an operation on elements in ProfContracts. We then define a function, profileOf, that takes an element in ProfContracts and returns an element in ProfileRep. If profileOf is applied to an element PC in ProfContracts, it will combine the default and individual contracts in PC. The element returned by profileOf will simply map entities to their contracts regardless of how the contract was specified (default or individual). We define conformance in terms of elements in ProfileRep. We also define a function, $\mathsf{legal}_{\mathsf{prof}}$, that determines legality of elements in ProfileRep.

First we define the function $\mathsf{profContracts}_{exp}$ that maps profile expressions into ProfContracts. A profile expression is mapped relative to an environment. The environment represents the context for the mapping. For example, if we are mapping a refinement expression, we need to access previously defined profiles.

$$E_p \in \mathsf{Env_p} = \mathrm{F}_{map}[\mathsf{Var}, \mathsf{ProfContracts}] \tag{74}$$

$$\mathsf{profContracts}_{exp} : (\mathsf{ProfExp} \times \mathsf{Env_p}) \rightarrow \mathsf{ProfContracts} \tag{75}$$

$$\begin{aligned}&\mathsf{profContracts}_{exp}(\textbf{profile } \{r_1 ; \ldots ; r_n\}, E_p) = \\ &\quad (\mathsf{indContracts}_{req}(r_1) \bullet \ldots \bullet \mathsf{indContracts}_{req}(r_n), \\ &\qquad \mathsf{defContracts}_{req}(r_1) \bullet \ldots \bullet \mathsf{defContracts}_{req}(r_n))\end{aligned} \tag{76}$$

$$\mathsf{profContracts}_{exp}(x_p, E) = E_p(x_p) \tag{77}$$

$$\begin{aligned}&\mathsf{profContracts}_{exp}(x_p \textbf{ refined by } \{r_1 ; \ldots ; r_n\}, E_p) = \\ &\quad E_p(x_p) \triangleleft (\mathsf{indContracts}_{req}(r_1) \bullet \ldots \bullet \mathsf{indContracts}_{req}(r_n), \\ &\qquad \mathsf{defContracts}_{req}(r_1) \bullet \ldots \bullet \mathsf{defContracts}_{req}(r_n))\end{aligned} \tag{78}$$

The function $\mathsf{indContracts}_{req}$ returns an element in IndContracts that represents the individual contracts of a requirements clause. The function $\mathsf{defContracts}_{req}$ returns an element in DefContracts that represents the default contracts of a requirements clause.

$$\mathsf{indContracts}_{req} : \mathsf{Require} \rightarrow \mathsf{IndContracts} \tag{79}$$

$$\mathsf{defContracts}_{req} : \mathsf{Require} \rightarrow \mathsf{DefContracts} \tag{80}$$

$$\mathsf{indContracts}_{req}(\textbf{require } c) = [] \tag{81}$$

$$\mathsf{indContracts}_{req}(\textbf{from } e \textbf{ require } c) = [(e, t) \mapsto \mathsf{contract}_{exp}(c)] \quad \text{where } c : t \tag{82}$$

$$\mathsf{defContracts}_{req}(\textbf{require } c) = [t \mapsto \mathsf{contract}_{exp}(c)] \quad \text{where } c : t \tag{83}$$

$$\mathsf{defContracts}_{req}(\textbf{from } e \textbf{ require } c) = [] \tag{84}$$

In the definition of $\mathsf{indContracts}_{req}$ and $\mathsf{defContracts}_{req}$ we use the fact that contracts have unique types in our type system. For a given requirements clause, we can derive a unique type for the contract

In equation (78), we describe refinement in terms of an operator, $\triangleleft$, on elements of ProfContracts. This operator is defined as follows:

$$\begin{aligned}&\text{for } (IP_1, DP_1), (IP_2, DP_2) \in \mathsf{ProfContracts} : \\ &(IP_1, DP_1) \triangleleft (IP_2, DP_2) - (IP_1 \triangleleft IP_2, DP_1 \triangleleft DP_2)\end{aligned} \tag{85}$$

We define refinement on the elements in ProfContracts as pair-wise refinement over default and individual contracts. To specify pair-wise refinement, we need to define the operator $\triangleleft$ on IndContracts and DefContracts. First, we define $\triangleleft$ on the domain DefContracts.

for $DP_1, DP_2 \in$ DefContracts with $t \in Dom(DP_1) \cup Dom(DP_2)$:

$$(DP_1 \triangleleft DP_2)(t) = \begin{cases} DP_1(t) & \text{if } t \in (Dom(D_1) \setminus Dom(D_2)) \\ DP_2(t) & \text{if } t \in (Dom(D_2) \setminus Dom(D_1)) \\ DP_1(t) \triangleleft_t DP_2(t) & \text{otherwise.} \end{cases} \tag{86}$$

If only one of the maps is defined on a particular contract type, then the mapping for that type is unchanged. If the maps overlap on a contract type t, then the mapping for t is defined in terms of contract refinement: we apply the maps to t, and do the refinement on the results of this application.

The definition of refinement in the IndContracts follows a similar pattern:

for $P_1, P_2 \subset$ ProfileRep and for $P_1, P_2 \in$ IndContracts with $(e,t) \in Dom(P_1) \cup Dom(P_2)$:

$$(P_1 \triangleleft P_2)(e,t) = \begin{cases} P_1(e,t) & \text{if } (e,t) \in (Dom(P_1) \setminus Dom(P_2)) \\ P_2(e,t) & \text{if } (e,t) \in (Dom(P_2) \setminus Dom(P_1)) \\ P_1(e,t) \triangleleft_t P_2(e,t) & \text{otherwise.} \end{cases} \tag{87}$$

Notice that this definition of refinement extends to the ProfileRep domain since the IndContracts and ProfileRep domains have identical structure.

The next step in our semantic definition is to define the function profileOf that takes an element in ProfContracts and combines the default and individual contracts into an element in ProfileRep. If the element in ProfContracts contains a default contract of type t, the returned element in ProfileRep will be a map that is defined on (e,t) for any entity e in the interface for which the profile was defined. Thus, profileOf is defined relative to a set of entities.

The set Entity is the set of all possible entity names; the set Interface is the set of all interfaces. We use *Ents* to refer to sets of entities, and we assume the existence of a mapping, called entities, that maps an interface to a set that contains all its entities.

$$e \in \text{Entity} \tag{88}$$

$$\text{Interface} \tag{89}$$

$$\mathit{Ents} \in \mathrm{P}_\omega[\text{Entity}] \tag{90}$$

$$\text{entities} : \text{Interface} \to \mathrm{P}_\omega[\text{Entity}] \tag{91}$$

We can now formally define the function profileOf as follows:

$$\mathsf{profileOf} : (\mathsf{ProfContracts} \times \mathrm{P}_\omega[\mathsf{Entity}]) \to \mathsf{ProfileRep} \tag{92}$$

$$\mathsf{extend} : (\mathsf{DefContracts} \times \mathrm{P}_\omega[\mathsf{Entity}]) \to \mathsf{ProfileRep} \tag{93}$$

$$\mathsf{profileOf}((IP, DP), Ents) = \mathsf{extend}(DP, Ents) \triangleleft IP \tag{94}$$

$$\mathsf{extend}(DP, Ents) = P \tag{95}$$

where

$$P(e,t) = DP(t) \quad \text{for } e \in Ents \text{ and } t \in Dom(DP) \tag{96}$$

The function extend takes as arguments an element in DefContracts and a set of entities. The element in DefContracts represents the default contracts of a profile. extend returns a profile in which entities are mapped to their default contracts.

As for contract declarations, we use a function called check to specify the legality of profile declarations. Check, in turn, uses a function $\mathsf{legal}_{\mathsf{prof}}$ to determine the legality of individual profiles. The function $\mathsf{legal}_{\mathsf{prof}}$ is defined on elements in ProfileRep—we cannot determine legality until the default and individual contracts have been combined, because the combination itself may be illegal.

$$\begin{aligned}&\mathsf{check}((x_p \textbf{ for } int = profExp; decl), E_p) = \\ &\quad \mathsf{legal}_{\mathsf{prof}}(\mathsf{profileOf}(\mathsf{profContracts}_{exp}(profExp, E_p)), \mathsf{entities}(int)) \ \wedge \\ &\qquad \mathsf{check}(decl, E_p \bullet [x_p \mapsto \mathsf{profContracts}_{exp}(profExp, E_p)])\end{aligned} \tag{97}$$

The function $\mathsf{legal}_{\mathsf{prof}}$ defines a profile to be legal if all its constituent contracts are legal.

$$\mathsf{legal}_{\mathsf{prof}} : \mathsf{ProfileRep} \to \mathsf{Bool} \tag{98}$$

$$\mathsf{legal}_{\mathsf{prof}}(P) = \forall\, (e,t) \in Dom(P) : \mathsf{legal}_{\mathsf{con}}(P(e,t)) \tag{99}$$

Having specified the rules for profile legaility, the following section specifies conformance for legal profiles.

6.6 Profile Conformance

Refinement is a static operation that provides notational convenience. For example, we use contract refinement to define a new profile in terms of an existing profile, and we specify only how the new profile is different (stronger than) the existing contract. In contrast, conformance is a dynamic operation that allows us to determine if a profile P_1 is stronger than a profile P_2.

Here we describe the semantics of profile conformance. We use the symbol $\prec$ to denote conformance; $P_1 \prec P_2$ means that the profile P_1 conforms to the profile P_2. The following rule gives a sufficient condition for conformance:

$$\frac{\begin{array}{l}\text{for } P_1, P_2 \in \mathsf{ProfileRep} : \\ Dom(P_2) \subseteq Dom(P_1) \\ \forall (e,t) \in Dom(P_2) : P_1(e,t) \prec P_2(e,t)\end{array}}{P_1 \prec P_2} \tag{100}$$

A profile P_1 conforms to a profile P_2 if P_1 specifies a contract for all the name and type pairs for which P_2 specifies a contract. Moreover, if the domains of P_1 and P_2 overlap, and (e,t) is an element in the domain intersection, then P_1 must map (e,t) to a contract that conforms to the contract to which P_2 maps (e,t).

We define profile conformance in terms of contract conformance. In the definition of contract conformance, we employ the notion of constraint conformance that was defined in section 6.4. Contract conformance is subject to the following rule:

$$\frac{\begin{array}{l}\text{for } C_1, C_2 \in \mathsf{ContractRep}: \\ Dom(C_2) \subseteq Dom(C_1) \\ \forall n \in Dom(C_2) : C_1(n) \prec_{(n,t)} C_2(n)\end{array}}{C_1 \prec C_2} \tag{101}$$

The domain of a contract is the set of dimension names for which the contract specifies a dimension value. As for profiles, $C_1 \prec C_2$ requires that the domain of C_2 is contained in the domain of C_1. Furthermore, the dimension constraints in C_1 must conform to the dimension constraints in C_2 for corresponding dimensions. The conformance rules for constraints are defined in section 6.4.

7. RELATED WORK

A fundamental concern of QML is to precisely specify properties of software components, namely their required or delivered QoS. In Section 7.1 we discuss some other approaches involving specification QoS and other aspects. The field of software metrics creates a theory for the formal representation of software properties. We compare our approach to the general theory of software metrics in Section 7.2. The section on software metrics relate to the domains that we use in our specifications. which we express specifications over these domains.

7.1 QoS Specification Mechanisms and Languages

Generally, interface definition languages, such as OMG IDL [Object Management Group 1995], specify functional properties, but lack any notion of QoS. Some interface definition languages do address various QoS aspects.

TINA ODL [Telecommunications Information Networking Consortium 1995] allows a programmer to associate QoS requirements with streams and operations. A major difference between TINA ODL and our approach is that TINA ODL include QoS requirements syntactically within interface definitions, and thus, 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 to support object-oriented specification. Neither does TINA ODL have any concept of confo

Similarly, Becker and Geihs [Becker and Gheis 1997] 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 for interfaces not individual operations. Their language allows a QoS specification to include any IDL type. They do not provide additional ways of specifying the sematics of

QoS attributes. Finally, they allow inheritance between QoS specifications but it is unclear which constraints they enforce to ensure conformance.

There are a number of languages that support QoS specification within a single QoS category. The SDL language [Leue 1995] has been extended to include specification of temporal aspects. The RTSynchronizer programming construct allows modular specification of real-time properties [Ren and Agha 1995]. In [Gupta and Pontelli 1997], a constraint logic formalism is used to specify real-time constraints.

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 [Ren et al. 1997], 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, but also enforce them. The approach in [Staehli et al. 1995] is to develop specifications of multimedia systems based on the separation of content, view, and quality. The specifications are expressed in *Z* [Spivey 1992]. The specifications are not executable per se, but they can be used to derive implementations. In [Blair et al. 1997], 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 [Campbell 1996] proposes pre-defined C-language *structs* that can be instantiated to QoS specifications for multi-media streams. The characterizations are limited to what is possible to express in the C-language; thus, there are notions of statistical distributions, etc. 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. It is clear, however, that there is a need for specification and statistical characterizations.

The approaches to multimedia QoS all assume that QoS constraints can be expressed in terms of the relative or absolute timing of events, such as sending and receiving messages. This assumption does not hold for QoS constraints in general. For example, it is not clear how to express a constraint over a service's failure masking property in terms of temporal constraints.

Zinky et al. [Zinky et al. 1997; 1995] presents 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.

The regions in QuO are similar to QML contracts in that they specify QoS levels. However, QuO does not contain a structuring concept that is similar to contract types and that captures the structure of QoS specifications within a particular QoS category. Moreover, QuO supports only numeric and measurable QoS dimensions. In contrast, QML allows the construction of user-defined domains. Finally QuO lack notions of refinement and conformance for QoS specifications.

Within the Object Management Group (OMG) there is an ongoing effort to specify whatever is required to extend CORBA [Object Management Group 1995] to support QoS-enabled applications. The current status of the OMG QoS effort is described in [Object Management Group 1997b], 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. ISO has an ongoing activity aimed at the definition of a reference model for QoS in open distributed systems. A recent working paper [ISO 1997] outlines how various dimensions such as delay and reliability could be characterized. It lacks, however, any proposal or recommendation for languages in which such constraints could be expressed.

There is an interesting similarity between formal functional specifications of interface semantics and formal QoS specifications. Specification matching is an important aspect of both types of specifications. In a recent paper, Zaremski and Wing [Zaremski and Wing 1997] propose a technique for specification matching with respect to semantics. Three of their main motivations for such a mechanism are:

—*Retrieval* : How can components be retrieved from software libraries based on semantic characterizations?

—*Reuse* : How can reusable components be adapted to fit a given subsystem?

—*Substitution* : How can we replace one component with another without changing the observable behavior?

The concerns with respect to QoS specification are very similar. For QoS we are concerned both with retrieving and reusing components with predictable QoS and with replacing existing components with new ones without degrading the QoS of a system. For QoS we are concerned with dynamic as well as static specification matching, while the work for semantics seems to focus on static design and implementation time checking. Nevertheless, there are obvious similarities thay suggest that a component really has three equally important aspects: syntax, semantics, and quality of service.

7.2 Software Metrics

Our discussion of software metrics is based on Fenton's book [Fenton 1991]. We are interested primarily in comparing our notion of domain to the concepts in [Fenton 1991]. Our intention is to create a context for our notion of domain. We also want to illustrate that capturing a particular software property, such as a QoS property, in terms of formal concepts is a general issue with a supporting theory.

We base our discussion on three main concepts from the theory of software metrics: *empirical relation systems*, *numerical relation systems*, and *scales*. We can

use an empirical relation system to characterize a software property. An empirical relation system is a set of entities and a set of relations on those entities. For example, the failure-masking property of a service could be characterized by an empirical relation system for which the entities are failure types. A relation on this set of entities could reflect how specific a failure type is. The response time of a service could be characterized by an empirical relation system for which the entities reflect response times. A relation for response times could be that one is better than another.

An important distinction between different empirical relation systems is how they can be mapped to a numerical relation system—a relation system where the entities are numbers. The *scale* of an empirical relation system determines how we can map it to a numerical relation system; the scale defines the allowed transformations (re scalings). Having fewer possible re scalings for a given relation system implies that the *scale type* is more restrictive. The theory of software metrics define five scale types: *nominal*, *ordinal*, *interval*, *ratio*, and *absolute*. Scales that are nominal are considered more restrictive than ordinal scales and so forth. We define the scale types in the following paragraphs.

If the set of allowed transformations consists of one-to-one mappings we say that the scale type is nominal. Systems for which entities are only labeled or classified are said to have nominal scales.

A scale type is said to be ordinal when the set of permissible transformations is the set of monotonically increasing functions. Thus, a transformation needs to preserve an ordering of the elements, but nothing more.

If we also require that the mapping function preserve the distance between entities, we say that the type of the scale is interval. Time and temperature are examples of attributes with interval scale types. As an example, think about rescaling the delay dimension from seconds to milliseconds. Such a rescaling needs to preserve both the ordering and the relative distance between values.

If the empirical system also includes a zero relation, the scale type is said to be a ratio. This is more restrictive than the interval scale type since it requires that we map a zero value to the numeric value 0. Length and delay are examples of attributes of ratio scale type.

Finally, absolute scale types restrict the mapping to the identity function; simple counting—such as a "number of failures" dimension—is an example of an attribute with an absolute scale type.

An important implication of scale types is that they determine the applicable statistical methods. For example, we cannot compute the mean of nominal or ordinal scale types. In addition, we can compute only the variance of ratio scales.

Domains in QML correspond to empirical relation systems. The names in a domain correspond to entities in an empirical relation system, and the "stronger-than" relation on a domain gives rise to a relation in the corresponding empirical relation system. A user-defined domain without an ordering has a nominal scale type. A user-defined domain with ordering has an ordinal scale type. Only the numeric domain has a notion of distance and a zero element, and is therefore both an interval and a ratio. Since statistical measures such as mean and variance require an interval scale type, statistical measures are available only for the numeric domain. Other characterizations such as frequency and percentile are more usable

more widely.

One challenge to writing accurate QoS specifications is to construct dimension domains that accurately capture the inherent nature of the QoS properties that we are trying to characterize. As we have illustrated, the process of domain construction is in many ways similar to the process of deriving software metrics.

8. CONCLUSION

To build systems that deliver predictable QoS, we need a way to precisely specify QoS properties for the system's components. We need QoS specifications to establish the non-functional boundaries between components so that we can reason about QoS correctness on a per-component basis. We also need QoS specifications to build QoS-enabled systems that can adapt to changes in their environments. Adaptation is necessary because the same system may be deployed in different environments and the conditions in those operating environments may change over time. For example, the same system may be deployed in both local-area networks and wide-area networks. Moreover, a system's environment can change dynamically due to variations in user workload and resource utilization. Finally, we need QoS specifications to establish client-server bindings in open systems in which services may come and go dynamically. In such systems, the binding process is dynamic and must take client QoS requirements into account to ensure that the service will satisfy those requirements. Thus, explicit specification of QoS necessary in order to deal with a variety of issues for QoS-enabled applications and systems.

We introduced QML as a language to specify QoS. We intend QML to be used for QoS specifications throughout the life-cycle of applications. We want to use QML at design time to establish QoS boundaries for components, and we want to use QML at runtime to facilitate adaption and QoS-based negotiation.

QML has a number of unique features. It allows multi-category QoS specifications: the same specification can contain statements about QoS properties in different categories, such as performance and availability. Multi-category specifications allow us to relate QoS requirements for multiple categories in one specification. QML also allows very fine-grained QoS specifications. For example, we can specify QoS properties for individual operations and attributes in interfaces. Finally, QML provides notions of both refinement, which allows incremental specification, and conformance, which allows runtime comparison of QoS specifications to determine if one satisfies the other.

QoS specifications at runtime must be first-class values that can be communicated in messages and bound to variables. We are currently working on a mapping from QML to both JAVA and C++ so that we can represent QML specifications as JAVA objects at runtime. We use CORBA IDL [Object Management Group 1995] as an intermediate step for this mapping to facilitate marshaling and demarshaling of QML specifications in CORBA-based distributed systems.

Besides implementing QML in various contexts, such as JAVA and CORBA, our future plans include QoS monitoring based on QML specifications. We plan to build monitors embedded in application infrastructures to provide runtime feedback on compliance of QML specifications. We envision monitors that are specific to contract types, but which can monitor compliance for all contracts that are instances of that contract type. For example, we envision having a generic performance mon-

itor that can monitor compliance of all possible performance contracts. Another direction that we are pursuing is to define and implement negotiation protocols that allow clients and services to reach agreement on the QoS for a specific client-server connection.

A. APPENDIX: CONCRETE SYNTAX DEFINITION

In this appendix we provide an extended backus-naur definition for the QML syntax. We will use *::=* for production definitions. $*$ and $+$ denotes zero or more and one or more occurrences, respectively. We use curly brackets ({}) for grouping and square brackets([]) for optional constructs. Non-terminal symbols are written within angle-brackets (<>). Semantic annotations are added by placing them in angle-brackets and underlining them (< *my name* >). Semantic annotations indicate a semantic aspect of a construct and should not be considered part of the production name. Terminal characters are written in quotes (';'). Reserved words are written in **bold** face and are also summarized in the table below:

contract	**type**	**profile**	**result of**
order	**with**	**set**	**order**
enum	**numeric**	**percentile**	**require**
decreasing	**increasing**	**variance**	**mean**
frequency	**refined by**		

Characters in QML should are coded according to the 8-bit coding standard ISO8859-1. Character sets and literals are defined as follows:

$$
\begin{array}{ll}
< letter > & ::= a|\ldots|z|A|\ldots|Z \\
< digit > & ::= 0|1|2|3|4|5|6|7|8|9 \\
< integer > & ::=< digit > + \\
< float > & ::=< digit >' .' digit * [\{'e'|'E'\}['+'|'-'] < digit > +] \\
< number > & ::=< integer > | < float > \\
< identifier > & ::=< letter > \{'\_'| < letter > | < digit >\}* \\
< name > & ::=< identifier > + \\
< percent > & ::= \%
\end{array}
$$

Next we define a context-free grammar for QML.

```
< declarations >     ::= < qmlDeclaration > *
< qmlDeclaration >   ::= < contrTypeDecl >
                     |   < contrDecl >
                     |   < profileDecl >
< contrTypeDecl >    ::= type < contract type name >'='< contractType >
< contractType >     ::= contract'{'< dimensionDecl > *'}'
< dimensionDecl >    ::= < dimension name >':'< dimensionType > [unit]';'
< dimensionType >    ::= < enumDef >
                     |   < setDef >
                     |   < numeric >
< orderSem >         ::= increasing
                     |   decreasing
< enumDef >          ::= enum '{'< nameList >'}'
                     |   < orderSem > enum '{'< nameList >'}'with < orderDef >
< setDef >           ::= set '{'< element nameList >'}'
                     |   < orderSem > set '{'< element nameList >'}'[with < orderDef >]
< orderDef >         ::= order '{'[< oneOrder > {','< oneOrder >}*]'}'
< oneOrder >         ::= < element name >'<'< element name >
< numeric >          ::= < orderSem > numeric
< unit >             ::= < unit name >
                     |   < unit name >' /' < unit name >
```

```
< contrDecl >        ::= contract '{'< dimConstraint > *'}'
< contrDeltaDecl >   ::= '{'< dimConstraint > *'}'
< contrExp >         ::= < type name >< contrDecl >';'
                     |   < contract name > refined by < contrDeltaDecl >';'
< dimConstraint >    ::= < simpleConstr >';'
                     |   < aspectConstr >';'
< simpleConstr >     ::= < dimension name >< numOp >< valueLiteral > [< unit >]
< valueLiteral >     ::= < number >
                     |   '{'< element nameList >'}'
                     |   < literal name >
< aspectConstr >     ::= < dimension name >  '{'< statConstr > *'}'
< statConstr >       ::= percentile  < number >< numOp >< valueLiteral >';'
                     |   frequency  < freqRange >< numOp >< number >< percent >';'
                     |   mean < numOp >< valueLiteral >';'
                     |   variance < numOp >< valueLiteral >';'
```

$< freqRange >$ ::= { '[' | '(' } $< valueLiteral >$ ',' $< valueLiteral >$ { ')' | ']' }
$< profileDecl >$ ::= $< \underline{profile}\ name >$ **for** $< \underline{interface}\ name >$ '=' $< profileExp >$
$< profileExp >$ ::= **profile** '{' $< reqClause >$ * '}'
| $< \underline{profile}\ name >$ **refined by** '{' $< reqClause >$ * '}'
$< reqClause >$ ::= [$< fromEntityList >$] **require** $< contractList >$ ';'
$< contractList >$ ::= $< contrExp >$ {',' $< contrExp >$}
$< fromEntityList >$ ::= **from** $< entityList >$
$< entityList >$ ::= $< entity >$ {',' $< entity >$}
$< entity >$ ::= $< \underline{operation}\ name >$
| $< \underline{operation}\ name >$ '.' $< \underline{argument}\ name >$
| **result of** $< \underline{operation}\ name >$
$< nameList >$ ::= $< name >$ {',' $< name >$}*
$< numOp >$ ::= '>=' | '>' | '<' | '<=' | '=='

ACKNOWLEDGEMENTS

The work presented in this paper has benefited greatly from interaction with and feedback from our colleagues in the Software Technology Laboratory in HP Labs. 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 also wish to acknowledge the feedback from Derek Coleman at Imperial College, Jørgen Nørgård at Beologic, Christian Becker at the University of Frankfurt; as well as the detailed comments from Patricia Markee.

REFERENCES

BECKER, C. R. AND GHEIS, K. 1997. Maqs—management for adaptive qos-enabled services. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services.*

BLAIR, G., BLAIR, L., AND STEFANI, J. B. 1997. A specification architecture for multimedia systems in open distributed processing. *Computer Networks and ISDN Systems 29.* Special Issue on Specification Architecture.

BOOCH, G., JACOBSON, I., AND RUMBAUGH, J. 1997. *Unified Modeling Language.* Rational Software Corporation. version 1.0.

CAMPBELL, A. T. 1996. A quality of service architecture. Ph.D. thesis.

FENTON, N. 1991. *Software Metrics: A Rigorous Approach.* Chapman-Hall.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

GUPTA, G. AND PONTELLI, E. 1997. A constraint-based approach for specification and verification of real-time systems. In *Proceedings of Real-Time Systems Symposium.* IEEE.

ISO. 1997. Working draft for open distributed processing—reference model—quality of service. Result from the SC21/WG7 Meeting.

KOISTINEN, J. 1997. Dimensions for reliability contracts in distributed object systems. Tech. Rep. HPL-97-119, Hewlett-Packard Laboratories. October.

LEUE, S. 1995. Specifying real-time requirements for sdl specifications—a temporal logic-based approach. In *Proceedings of the Fifteenth IFIP WG6 (Protocol Specification, Testing, and Verification XV).*

Object Management Group 1995. *The Common Object Request Broker: architecture and specification*, revision 2.0 ed. Object Management Group.

Object Management Group 1997a. *CORBA Services — Trader Service*, formal/97-07-26 ed. Object Management Group.

Object Management Group 1997b. *Quality of Service: OMG Green paper*, Draft revision 0.4a ed. Object Management Group.

REIBMAN, A. L. AND VEERARAGHAVAN, M. 1991. Reliability modeling: An overview for system designers. *IEEE Computer*.

REN, S. AND AGHA, G. 1995. Rtsynchronizer: Language support for real-time specifications in distributed systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*. ACM.

REN, S., VENKATASUBRAMANIAN, N., AND AGHA, G. 1997. Formalizing multimedia qos constraints using actors. In *Proceedings of the Second IFIP International Conference on Formal Methods for Open, Object-Based Distributed Systems*.

SPIVEY, J. M. 1992. *The Z Notation: A Reference Manual*, Second edition ed. Prentice-Hall.

STAEHLI, R., WALPOLE, J., AND MAIER, D. 1995. Quality of service specification for multimedia presentations. *Multimedia Systems 3*, 5/6 (November).

Telecommunications Information Networking Consortium 1995. *TINA Object Definition Language*. Telecommunications Information Networking Consortium.

TENNENT, R. D. 1991. *Semantics of Programming Languages*. Prentice-Hall.

ZAREMSKI, A. M. AND WING, J. M. 1997. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology 6*, 4 (October).

ZINKY, J. A., BAKKEN, D. E., AND SCHANTZ, R. D. 1995. Overview of quality of service for distributed objects. In *Proceedings of the Fifth IEEE conference on Dual Use*.

ZINKY, J. A., BAKKEN, D. E., AND SCHANTZ, R. D. 1997. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems 3*, 1.