# Some Requirements for Quantitative Annotations of Software Designs

Dorina C. Petriu and Murray Woodside
Dept. of Systems and Computer Engineering,
Carleton University, Ottawa Canada
{cmw | petriu} @ sce.carleton.ca

## Abstract

Various initiatives, including the RFP for MARTE [6], call for annotations that define quantitative measures and values to be added to software designs. In MARTE these annotations will indicate timing and memory-use properties of the software and of its behaviour, as well as timing, capacity and utilization properties of resources. Other kinds of non-functional requirement analysis need different types of properties; for instance reliability analysis calls for properties such as failure rates and probabilities. This paper defines general requirements for quantitative annotations, related to their function in evaluating non-functional requirements of software specifications and to their usability. It considers how two previous profiles, SPT [4] and QOS [5], address these requirements, and raises issues and questions related to defining a profile for MARTE [6].

## 1. Introduction

The background of this work is the two UML Profiles for Schedulability, Performance and Time (SPT) [4] and for Quality of Service (QOS) [5], which are concerned with adding quantitative attributes to elements of a software specification in UML. There are also other descriptions of profiles; for instance, a reliability profile is proposed in [1]. Annotating a specification is different from other problems, such as transforming an annotated UML model into an analysis model, or evaluating the analysis model [7].

This paper considers general requirements for quantitative annotations, with particular reference to the authors' area of performance evaluation. The purpose of annotations is to support the specification of non-functional requirements, and also their evaluation by analysis tools. Since the annotations must be usable by software designers and must also support the concepts of the analysis models, they must bridge the gap between the domains of software specification and evaluation.

We consider that the requirements can be grouped as follows:

- What quantitative characteristics/properties should be considered, and more generally how they are to be defined. This is rooted in the evaluation (analysis) domain, but must be understood by the software designer.
- How particular instances of the quantitative characteristics are to be attached to model entities in the UML specification. This is rooted in the software

specification domain, but must attach attributes correctly and with sufficient expressive power, for the analysis.

- How relationships between different quantitative properties are to be defined. This is relevant to the evaluation domain.
- Expression of constraints on or between the quantitative properties, which may express requirements on the system. This is relevant to the evaluation domain.
- Across the preceding four groups of requirements, usability of the annotations implies that the effort by the software designer should not be excessive, and the definitions and usage should be consistent among themselves.

The scope of this paper is to collect together a set of requirements, not necessarily exhaustive, based on our experience in software performance and in applying the SPT profile; to discuss how the SPT and QOS profiles succeed or fail to meet them; and to formulate the resulting issues for further discussion, in the further development of this field in MARTE [6].

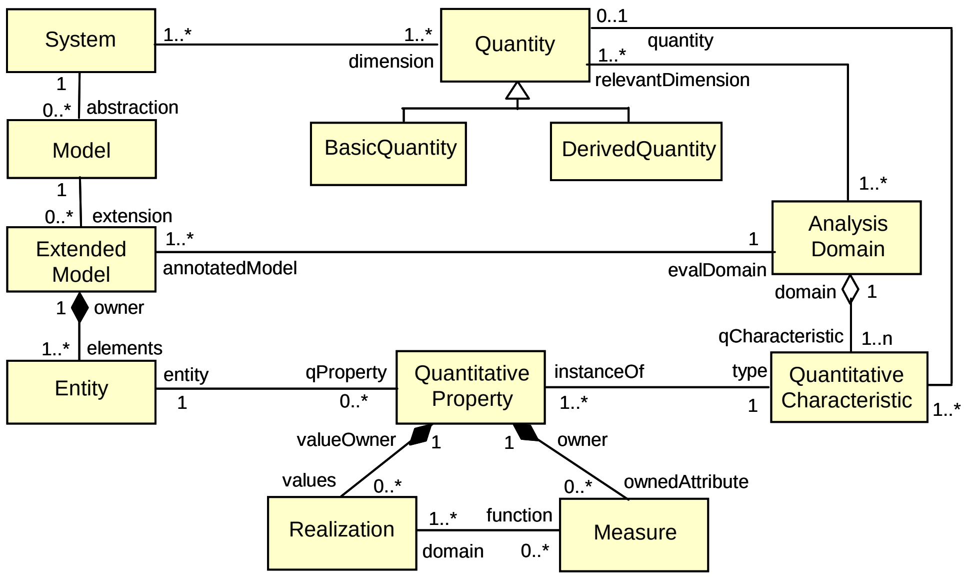


**Figure 1.** Metamodel for quantitative characteristics and properties

According to measurement theory, physical Systems are characterized along different dimensions that correspond to a set of measurement Quantities, which can be Basic or Derived. The most used Basic Quantities are length, mass, time, current, temperature and luminous intensity. The units of measure for the basic quantities are organized in systems of measures, such as the universally accepted Système International (SI) or International System of Units. Values expressed in the same unit can be compared. Derived Quantities (e.g., area, volume, force, frequency, etc.) are obtained from the Basic Quantities by known formulas.

Software Systems can be analysed in many different ways. Each Analysis Domain uses a Model of the System, which is an abstract representation that focuses on certain system characteristics and ignore others. Due to the abstraction, only some Quantities are relevant to a certain Analysis Domain. For instance, in the domain of performance analysis of software systems, the relevant Basic Quantities are time and sometime length (expressed in units of memory occupancy), whereas mass, current, temperature and luminous intensity are irrelevant. However, some of these may become relevant if the performance analysis is extended to include, for example, power demands.

An Analysis Domain uses a set of Quantitative Characteristics (or *q-characteristics* in short), which establish the ontology of the domain. For instance, in the case of software perfomance analysis the q-characteristics are throughput, response time, utilization, CPU execution demand, etc. These are at the same abstraction level as the QoSCharacteristics from the QOS Profile [5]. A q-characteristic refers to a Quantity and can be expressed by using a data type that describes its unit and other properties, as discussed later in the paper.

A Model (which is considered here to be expressed in UML) can be extended by standard UML mechanism with additional semantic expressing concepts from a certain evaluation domain. An Extended Model contains Entities, which are extended model elements that represent concepts from the analysis domain. For example, some typical performance-related Entity types are: *Step* (an execution block as defined in SPT), *Scenario* (a sequence of Steps), *Resource* (as defined in the General Resource Model [4]), *Service* (an operation offered by a Resource or by a component of some kind; it may be further defined by a Scenario). For our purposes, an Entity is an extended model element that may be characterized by certain Quantitative Properties (or *q-properties* in short), which in turn are instances of different Quantitative Characteristics. For each Entity type, a set of Quantitative Characteristics can be defined in a given Analysis Domain. The q-properties are specified by the designer within the UML model, and attached to design entities. Examples are: the total delay of a Step when executed (including queueing delays), the utilization of a resource, the response time and throughput of a Service, etc.

When the system is simulated or executed, a set of values may occur for each q-property; these are named Realizations. Different Measures may be used to characterize a given q-property. A Measure is a (statistical) function (e.g, mean, max, min, median, variance, standard deviation, etc.) applied to the set of Realizations of the respective q-property. Depending on the evaluation method used, some of these q-properties are required as input for the evaluation (e.g., CPU execution demand, arrival rate), whereas others are obtained as output (results) of the evaluation (e.g., response time, utilization).

## 2. Quantitative Characteristics

Each evaluation domain (performance, schedulability, dependability, etc) employs a set of domain Quantitative Characteristics (*q-characteristics* in short), which represent types of properties describing computer systems in general and the respective analysis domain in particular. Q-characteristics include both *input q-characteristics* of the software, its configuration and its workload needed for the evaluation, as well as *output* *q-characteristics* to be evaluated during the analysis. For performance analysis, the evaluation requires input q-characteristics such as the CPU demand of an operation or the number of processors in a node, and gives *output q-characteristics* such as response time and utilization. A q-characteristic is associated with a Quantity (be it Basic or Derived, from Fig. 1), such as time or frequency, which determines its unit.

The definition of these q-characteristics establishes the ontology of the evaluation, selected from the ontology of the evaluation domain. In many domains there are q-characteristics whose names and definitions are universally agreed (e.g. throughput), others whose definition is generally accepted but has multiple semantic variants according to different authors (e.g. execution time of a function), and still others which require a definition whenever they are used (e.g. success rate of an operation). To expand on these examples from the performance domain, throughput is always a rate of occurrence of some event, such as the completion of some operation, in operations per unit time. The operation must be identified, but the concept of throughput is universal. Execution time, however, may have different meanings: a) the CPU time used by the function, or b) the total time it takes to complete the function, including waiting for the CPU, blocked for I/O, etc. Finally, a success rate always requires a definition of what constitutes success, which is model-dependent. Sometimes, success is identified by taking a particular path in the execution, in other cases it may be a particular post-condition or a property of the entire execution path within the operation.

Usability suggests the merit of defining a set of standard q-characteristics for a domain, so they can be referred to easily and, consequently, every user of the annotations means the same thing. For q-characteristics with well-known variants, a set of definitions can be standardized, which cover the important cases with differently-named measures, and these can be translated if necessary by domain specialists for the use of an analysis tool with different names. However the third class of q-characteristics whose meaning is model-dependent, requires a capability for users to define their own q-characteristics; This also makes the annotations more flexible for adding new “standard” q-characteristics definitions over time.

Thus flexibility and expressive power requires that the users have the capability to define their own quantitative measures, but usability requires a set of standard measures that can be used in straightforward way.

The q-characteristics discussed here correspond to the QoSCharacteristics from the QoS Profile [5]. However, we prefer the prefix “*quantitative*” instead of “*QoS*” for the following reason. In general, the concept “Quality of Service” is used in relation with the user’s satisfaction with the performance of the system, and therefore seems to suggest that *QoS Characteristics* describe those end-to-end performance measures that are perceived by the user (such as response time). However, the q-characteristics we consider here include all kind of quantitative measure describing the internals and externals of the system, many of which are not directly related to the user and his/her perception of the system performance. Therefore, we propose to use the prefix “*Q*” from “*quantitative*” instead of “*QoS*” for the UML stereotypes that will represent these concepts in MARTE (e.g., `<<QCharacteristic>>, <<QProperty>>, <<QValue>>`, etc.)

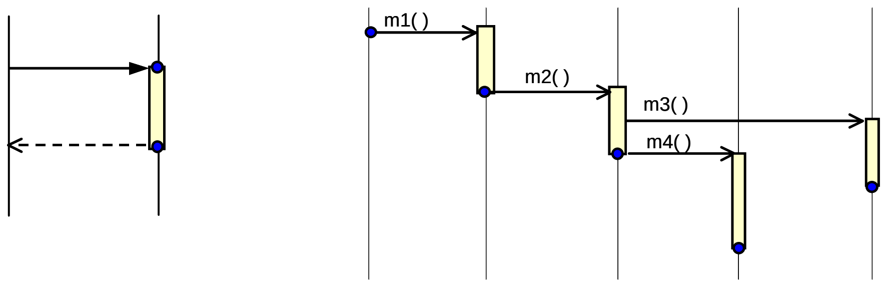
## 3. Quantitative Properties

As shown in Figure 1, Quantitative Properties (*q-property* in short) are Quantitative Characteristic instances that are associated with Entities (model elements) in order to represent their domain-specific properties. The way a q-property is declared and attached to entities in a UML diagram defines its semantics for the analysis. Even though this paper does not propose a concrete solution for attaching q-properties to UML model elements (entities), the following discussion considers the attaching mechanisms previously used in the SPT and QOS Profiles: a) notes containing stereotypes and tagged values attached to concrete model elements; b) constraints attached to model elements. In general, is it possible to *directly* attach annotations only to those model elements that meet one of the following two conditions: a) are represented in the visual notation for UML 2.0 diagrams, or b) have a name.

We can distinguish *primary* q-properties, which can be identified directly with diagram entities, and *secondary* q-properties, which must be derived from the primary ones by computational expressions.

### *3.1 Examples for Performance: Quantitative Properties of Behaviour*

An example of a primary q-property that can be easily attached to a UML diagram is the total delay of an operation represented by a UML model element such as an Activity or ExecutionOccurrence [3]. An example for the latter is shown in Fig.2.a: the delay annotation can be directly attached to the visual representation of the corresponding ExecutionOccurence, with the implicit meaning that it lasts from its beginning event to its end event.



***a) Delay for an operation execution***
*- can be attached to the ExecutionOccurrence*

***b) Delays for complex combinations of events***
*- a progress point may be defined by reaching an event*
*- each progress point has its own delay from the start*

**Figure 2.** Point-to-point delay interval

Another example of a primary q-property is the duration of an interval from one event in one lifeline, to a second event in another lifeline (which we shall call a “point-to-point delay interval”), as in Figure 2.b. However, attaching such a delay annotation is hampered by the fact that the visual notation for sequence diagrams does not allow the users to draw or to attach names to the events shown by small circles in Figure 2 (even though such events are represented in the UML 2 metamodel [3]). So, it is not possible to attach annotations to events directly on the diagram, but it is possible to specify an interval between two events by referring to other visualized model elements (such as messages) that are related to those events. We could specify an interval as follows:

```
m1.sendEvent - m4.sendEvent
```

It is interesting to note that specifying an interval as the difference between an ending and a starting event does not necessarily implies that its measures (such as mean, max, etc) will be computed by subtracting the time stamps of the two events. (See section 3.3 for a more detailed discussion on measures and realizations).

An example of a secondary q-property type is the fraction of processor utilization imposed by a particular operation (as opposed to the whole processor utilization), defined as the quotient:

*CPU_utilization_by_an_operation = operation_throughput / operation_CPU_demand*

Such a q-property is useful when analyzing the performance of a system, as it helps to identify the most expensive operations that may have to be optimized. However, it is not easy to illustrate its meaning through UML diagrams.

For performance evaluation, some "standard" q-property types are easily identified, and these could be supported by a shorthand notation (for usability). They include:

- delay properties of execution of an operation or service: *total delay* from request arrival to completion, *queueing* time before execution, *total time to execute* including queueing, total *operation blocking* time on nested operations, *CPU waiting* time, *CPU execution time*. These are all intervals, and any measures of these intervals may be necessary for performance analysis.
- any measure of delay of a message from sending to receiving;
- throughput (i.e., mean frequency of execution) of any operation, message, or behavioural entity;
- repetition interval of any operation, message or other behavioural entity, which gives more detail on the repetition (the mean interval is *1/throughput*, but the interval has statistical properties as well);
- service time of an operation provided by a resource which requires in turn other resources.

The above q-property types all may be the "outputs" (results) of an evaluation. Other types may be the "inputs", for instance:

- CPU demand of an operation (should equal CPU execution time, but the former is a preliminary estimate and the latter is an observation);
- service times of operations by some resources (known in advance, such as a disk device);
- arrival rate of requests from the environment (a throughput);

- multiplicity of a resource, such as processors at a node, threads of a process, buffers in a pool, initial value of a counting semaphore, multiprogramming limit of a node (a count);
- number of users of a system with a closed workload, or (equivalently) the concurrency limit of the load source (a count);
- external delay of users with a closed workload;
- queue length threshold, e.g. for diverting overflow traffic (a count).

Some of these q-properties may be either inputs or outputs depending on the context, for example:

- a message latency may be known as an input, or it may be determined by evaluation;
- an arrival rate may be given (open system) or derived given the number and external delay of users (closed system);
- the service time of an operation by a resource (defining its QoS) appears in both lists.

## *3.2. Examples for Performance: Quantitative Properties of Objects*

Objects (e.g. resources) may also have q-properties, such as utilization of a resource or size of an object. In particular, in both SPT and QOS it is stated that resources have QoS attributes. However it is more strictly true that services offered by resources have QoS attributes, and we propose that all QoS attributes should be formally attached to *services* rather than to the resource that offers them. (Only if a resource offers a single service, then we could associate the QoS with the resource without ambiguity). If a resource is an object, then a service is an operation carried out by the object, which is usually indicated by a method or an interface. For components, the operations are detailed within the interface definition, and details of the service may be defined by behaviour within the component.

Q-properties of objects include:

- utilization of a resource, or partial utilization of a resource by operations of a particular service;
- delay of a service, which gives the various QoS characteristics as measures of the delay. This can include delay to create or destroy the object;
- throughput of all services of the object;
- size of the object (memory units);
- mean time to failure, time to repair, etc.

## *3.3. Realizations and Measures of Q-Properties*

Realizations (see Figure 1) signify values that occur during the execution of the system (for instance, measurements on a simulation or test). A q-property may be realized once, or its realization may be a series of values over an extended run. In a cyclic deterministic system in which each cycle has the same values, a single realization is sufficient. In performance analysis with random traffic, a long run may be necessary with long sequences of values in order to obtain accurate evaluation results.

Measures (see Figure 1) are basically functions of Realizations that express the properties of interest for the evaluation, for example the mean value, the maximum value, the variance, etc. They may also be functions of a probability distribution for the realizations, based on an analysis model. For example, a measure might be the expectation or a theoretical bound of a q-property. Since Measures are owned by a q-property, they can be thought of as q-property attributes.

The associations between QuantitativeProperty, Realization and Measure (see Fig.1) are navigated in different ways depending on the type of evaluation (e.g., measurement of a system implementation, simulation, analytical solution of a performance model). Another factor is whether the respective q-property is an input or an output.

The difference stems from the fact that realizations for q-properties exist only in cases where the analysis is done by executing the system/model, either by testing/measurement or by simulation. A set of realizations is obtained in such cases for each output q-property; the measures can be obtained by applying the respective (statistics) functions to the set of realizations. The quantitative results obtained for the measures are reported as attributes of the respective q-property. In the case of analytical models, however, realizations are never computed; the output measures are calculated instead by analytical methods from the model parameters. The situation is different for input q-properties, which are given quantitative values or are described by a distribution with different parameters through the annotations, as discussed below.

In any case, even though Realizations may exist, they are not annotated directly on the UML model (too much information!) They are represented instead in an abstracted way through the corresponding Measures, which must be annotated on the UML model.

### *3.4. Source of Quantitative Annotations*

It is a peculiarity of these quantitative properties that the same property may be defined separately from different sources. An obvious example is *required* values, versus *achieved* values, but additional subdivisions may arise. For example the achieved value may be measured in a certain test (there may be more than one of these for the same q-attribute), or be estimated by an analytic model. Values may be stated for different execution environments. Input attributes may take assumed values based on the expertise of the designer/analyst, and there may be more than one of these (e.g., for worst-case and best-case, or representing the expertise of different parties). The ability to designate different sources and to compare the values given by different sources is fundamental to the full exploitation of evaluation methodology.

Again there appear to be "standard" sources, at least for performance and schedulability analysis, including:

- required value
- measured value, with a string to indicate details (measurement experiment, platform...)

- assumed value, with a string for details (platform, workload case...)
- estimated value, with a string for details (platform, workload case...)

SPT uses a "source-qualifier" string to differentiate these, with cases for *required*, *assumed*, *measured* and *estimated*. It would be desirable to support user-definable sources, apart from the strings described to convey details; perhaps just a string would be enough for this. However for tool support it seems desirable to support standard codes for required and achieved values. Clearly it should be possible to define as many versions of a single q-property, from different sources, as necessary. The capability for defining details could be used to list the results of a series of tests or model analyses representing different platforms, or different imposed load levels.

The purpose of expressing different sources is to gather the maximum information from the designer side. Automated analysis tools will have to filter the values according to the cases of data needed for the current analysis.

## 4. Requirements for Attachment of Q-properties to UML Models

"Annotation" is a process of attaching information to selected UML model entities. These model entities have UML types, but for annotation purposes they also have "domain analysis types" which we shall assume are assigned by stereotyping the model entities. An example is an ExecutionOccurrence which is stereotyped as <<Step>>; from the analysis point of view we will call it an analysis Entity of type Step. We must be able to annotate structural entities such as objects and nodes, as well as behavioral entities such as lifelines, execution-occurrences, messages, activities and transitions.

We identified the following requirements for attaching QuantitativeProperties to Entities:

1. Each Entity type has a certain set of possible types of q-properties (i.e., Quantitative Characteristics). For example a Resource should be characterized by its utilization and throughput; a Step by its CPU demand and total delay. This set should be user-extensible, to accommodate user-defined q-properties.

   Notice that these sets are not a partition of the Quantitative Characteristics; the same characteristic may be applicable to different Entity types. Thus throughput may be a property of a Resource but also of a Scenario or a Step. On the other hand some may depend on sub-typing, so processing rate or clock speed is a property of a processingResource but not of a logical resource such as a semaphore or process.

2. In the majority of the cases, a q-property is attached to a single Entity, but in some special case it may be attached to two (e.g., a delay between two events). For usability reasons, it would be better to consider that each q-property belongs to a single Entity (in other words, it is an attribute of the Entity). However, the chosen approach should compromise the ability to attach a q-property to more than one Entity when necessary.

3. Each Entity instance has its own values for q-properties. If q-properties can be applied to a class, they could be interpreted as default values for its instances. Thus the attachment mechanism must allow for attaching q-property values to any Entity instances, which means attachment to any UML model entities.

4. An Entity instance may have more than one value specified for a given q-property, with different sources, which are defined as part of the value. Thus a given Resource may have a required utilization, an utilization estimated by analysis, and two different measured values taken under different conditions.
5. In order to deal with complex systems with replicated structures and collections of identical Entity instances, it should be possible to define the properties once for the members of a collection (but not only once for all instances of an Entity class).
6. There are common *global quantitative parameters* that may affect many other q-properties through dependencies, which in turn can be expressed through functional relationships as described in section 5. These global parameters need to be attached to the analysis as a whole, either at the level of a UML diagram or at the level of a collection of diagrams. Examples of global parameters include the following types:
   - global attributes of the software application itself, such as the size of a database, length of a list, etc., which affects the processing time of many operations inside the application (e.g., database search, list sort, etc.);
   - attributes of the platform, such as file operation costs, message overheads, middleware costs, or operating system operation costs. While it may be possible to attach these properties to a platform model, in many cases the platform will not be described in enough detail, so it would be desirable to use instead some global variables that characterize the platform at a high-level of abstraction.
   - different cases of the evaluation, which are in effect points of variation for the evaluation. This could include different platforms, different kinds of users, or (for systems with replicated components) different system scales.

## 5. Expressing Quantitative Values

Concrete Quantitative Values need to be assigned sometimes to q-properties, other times to their attributes (such as measures). We shall consider here only the definition of these quantitative values, and not the way they are used in evaluating a system. The traditional output q-properties that describe performance use the following quantitative values:

- *time instant* of an event, which is the interval from a defined reference event to this event
- *time interval*, the time from one event to another, (which can be applied to all kinds of delays)
- throughput, or *frequency* of an event or a class of events (a derived quantity = 1/time)
- *probability* or fraction (a dimensionless ratio of throughputs or other quantities)
- *indicator* signifying true/false, yes/no or success/failure.

In general, the quantitative inputs giving the parameters required for performance analysis include:

- CPU demand in terms of *time* (the CPU time required to execute some operation)
- CPU demand in terms of *CPU operations* or *cycles* (a *count*) - the same idea as above, but different units; this is more fundamental but is used less often
- *probability* of a particular branch or alternative in the behaviour specification,
- loop *count* in the behaviour specification
- *frequency* of arrivals (a throughput)
- *count* of the numbers of resources such as CPUs, threads, buffers, or number of users of a system
- *size* (in memory units) of a storage resource or an object.

These quantitative values are very general. In a suitable context, they can be used to define all kinds of delays (waiting times, latencies, periods of event streams, laxity, deadlines), all kinds of frequencies, probabilities of conditional behaviours and of success in meeting delay targets, thresholds for scheduling strategies in terms of delay or queue size, or indicators for properties such as schedulability. They also apply to other quantitative evaluation domains, such as reliability.

The elements required to specify a quantitative value are:

- a value type,
- a quantitative value specification (a constant value, variable or expression)
- a direction plus/minus (which way is “better”, if that is relevant)
- units
- constraints on values.

***The value type*** is more complex than it might seem. Some quantities are intrinsically real, some are integers. Even for integer quantities, a measure such as a mean value (e.g. the mean number of busy processors) is real. This suggests using the value type *real* for all quantitative annotations, with integer values as a special case of real. This has been found to be practical in several programming languages, and would simplify the definitions of data types (and enhance their usability). The requirements for a value type appear to lead towards a kind of union data type, similar to the one defined for the *RTtimeValue* type in SPT.

Further, the value type must be able to express different *measures* of the same q-property, which express different forms of knowledge about its value. For example, the results of evaluation could be presented alternatively as a mean, a mean and variance, or a histogram. There is again a set of “standard” ways of recording measures, which can be listed and included with a common syntax understood by all users of the annotations, and also the possibility of exceptional definitions by the user. Some of the “standard” measures are:

- a point value
- a minimum value, a maximum value, or a range (minimum, maximum)

- a mean value, or variance, or *i*th moment, or *i*th central moment
- a list of the first *n* moments or central moments
- a distribution with its parameters (taken from a list of standard distributions found in textbooks, such as `exponential(mean), Erlang(mean, kernel), uniform(min, max)`).
- a histogram with equal cells: minimum boundary, cell size, number of cells, array of counts including underflow and overflow.
- the probability of exceeding a stated threshold or target value: a pair (value, probability)
- a histogram with unequal cells: array of cell boundaries, array of counts.

It would be desirable if the syntax for values included some self-describing indicator for the measure used, if it is a standard measure.

***Quantitative Value Specification.*** A quantitative value can be specified either as a "constant" value (QValue), as a variable (QVariable) or as an expression (QExpression), as shown in Fig.3.

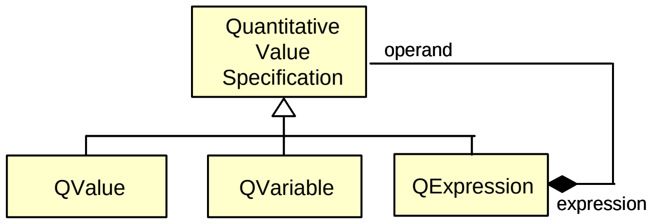


**Figure 3**. Metamodel for Quantitative Value Specification

***QValue*** represents a quantity of a given value type. It corresponds to the QoSValue from the QOS Profile [5]. The expression of a quantity should express the tuple:

*quantity = [direction, value_specification, units].*

***QVariable*** is required for situations where the concrete value is not yet known when specifying the software. QVariable names can also be used as placeholders for results in the UML annotations, while using the same name within the analysis domain. QVariable names thus help to bridge the gap between the UML specification and the analysis domain. QVariable names are also needed to support quantitative relationship between different q-properties, as discussed later. SPT used the syntax "*$string*" for names of quantitative variables, to distinguish them from names used in the UML model itself.

Names raise the question of scope. It should be possible to combine diagrams created separately, into a single analysis, where the same name may have been used more than once. Some way to disambiguate these names is necessary, and it should also handle the problem of UML models that are simultaneously annotated for multiple kinds of analysis. Let us consider for instance, that an annotated diagram has a stereotype

`<<ThisAnalysis>>` with tagged value `{diagram = "ThisBehaviour"}.` A quantitative variable `$ResponseTime` could be referenced unambiguously as `ThisAnalysis.ThisBehaviour.$ResponseTime.`

The reference would not be based on names of UML objects, but names of the annotations.

***QExpression.*** In all quantitative evaluations there are occasions where some quantities are derived from other quantities. This is so basic to quantitative studies that it must be provided in the annotations discussed here. As a motivating example, suppose there is a characteristic size (call it *$dataSize*, in bytes) of a data structure that is stored, retrieved, processed and passed in messages. The CPU cost of operations, the delay for transmitting messages, the memory space required for storage, are all functions of *$dataSize*. It is much easier as well as more informative, to define these quantitative properties by expressions; also, the evaluation is more robust to changes in the design or the usage of the system, that could change the value of *$dataSize*. We can call *$dataSize* an *independent parameter* of the evaluation.

Examples of QEpressions related to performance analysis are:

- CPU demand to process a message = *constant + variable_cost * message_size*
- CPU demand to sort a list = *constant * log (list_length)*
- number of disk operations = *file size / block size*

In [Woodside2001] these kind of functions were described as "resource functions", and technology to estimate different resource functions from run-time measurements was described.

Similarly, for schedulability analysis, the execution times of various signal-processing operations are given by well-known functions of the length of a block of data.

From an evaluation point of view, establishing these relationships makes the analysis more flexible and also more robust. Evaluations are often carried out over ranges of values of the independent parameter, covering cases of interest; it is easier to express the ranges only for the independent parameters and to let the dependencies calculate the rest.

It is reasonable to permit q-properties to be defined as follows:

(1) a quantitative values of the appropriate type

(2) a quantitative variable (no value given)

(3) a quantitative expression in terms of independent parameters

(4) a combination of (2) and (3), where the variable name can be used in analysis reports or in defining a further dependency.

Thus the mean or max CPU demand property of an operation might be expressed (using the *$string* notation for variables in suitable units) as:

(1) *17.3*

(2) *$opDemand*

(3) *$a + $b * $dataSize*

(4) *$opDemand = $a + $b * $dataSize*

What is said above for input parameters for the evaluation also applies to outputs; a derived output q-property may be defined as a function of other q-properties. For instance a throughput in messages/sec is related to throughput in bytes/sec by the message size, or a total throughput is the sum of throughputs for all different kinds of simultaneous users.

The requirement to annotate with variable names and expressions is a significant intrusion of the evaluation domain into the specification. Some support for these concepts seems however to be absolutely essential. A minimum would be to specify only variable names for quantitative annotations (as in the form (2) of value listed above), and to restrict all functional relationships in the analysis domain. The option of allowing only numeric values for q-properties, however, is insufficient for any quantitative evaluation.

***Units*** are an attribute of most quantities, and it is important that standard forms be used. If seconds can be specified by “seconds”, “sec” and “s” then automated translators may err in setting up the parameters of the analysis model. International standards are available for units, but may not meet usability criteria. In tools, pull-down menu selection can solve this problem. Notice that even if a measure is a ratio (i.e., dimensionless) it may be necessary to state whether it is a fraction or a percentage. For integer counts of resources, users, queues, the units of “count” imply positive integers.

It might be preferred by some users to have a single set of units for all variables (all times in seconds, all ratios in fractions, for instance), with units set for the whole analysis and each quantity only identifying “timeUnit” or “ratioUnit”. In other cases it might be more convenient to state some delays in micro-seconds and others in seconds. Uniform units could be an option, amounting to user-defined units declared for the analysis as a whole.

***Summary.*** To summarize the definition of a q-property must express the tuple:

*q-property = [ attribute-name, source, quantitative_value ]*

Some of the discussion above is specific for performance, but much of it generalizes easily to other evaluation domains.

## 6. Expressing Constraints

Constraints can be defined both for Quantitative Characteristics (at class-level) and for Quantitative Properties (at instance-level). Quantitative Characteristics constraints stem from the intrinsic nature of the respective measure (such as delay >= 0, percentage within [0, 100], utilization within [0, 1.0]).

Quantitative Properties constraints may be used either to specify the input values for performance analysis, or to express requirements on the analysis results. Equality constraints (invariant) may be used for giving the input values, as for example:

*stepname.cpuDemand = 5 ms*

*stepName.cpuDemand = (3.2 + 4.1 * $Size) ms*

Inequality constraints can be used to specify requirements on output values, such as:

*scenarioName.totalDelay* < *scenarioName.requiredDelay*).

A user-defined constraint is a condition in a specified language, whose syntax and interpretation is the responsibility of the tool [3]. Although OCL is preferred for writing constraints, other languages are also allowed (programming or natural). Constraints will have to be able to accept not only values and names from the UML domain, but also quantitative values (with source and units), quantitative variables and expressions specified in annotations, as discussed in the previous section.

## 7. SPT and QOS Profiles

The following table summarizes how the SPT and QOS Profiles meet the requirements for quantitative annotations listed in the paper.

| **Requirement** | **SPT Profile** | **QoS Profile** |
|---|---|---|
| Annotation process | Light-weight | Heavy-weight |
| Allows for user-defined measures | No (measures are predefined) | Yes (targeted for user-defined measures) |
| Type for time values | RTtimeValue | No |
| User-defined delay measure between an arbitrary pair of events | No | No |
| Quantitative variables and independent global parameters | Yes<br>Part of the TVL language | No |
| Expressions for defining quantitative properties | Yes<br>Part of the TVL language | No |
| Quantitative variables and independent global parameters | Yes<br>Part of the TVL language | No |
| Expressions for defining constraints | Limited | Yes<br>Full power of OCL |

## 8. Conclusions

This paper defines a metamodel for Quantitative Characteristics and Properties necessary for different kinds of quantitative analyses, with focus on the performance analysis domain. It explains the nature of Realizations and Measures and identifies their relationships with Quantitative Properties. The relationships between quantitative annotations and UML model Entities are discussed; such relationships span the gap between the software modeling domain and the analysis domain. Based on the proposed metamodel, a list of requirements for attaching quantitative annotations to UML model elements is established. A reach set of examples regarding the nature and the meaning of different performance quantities is given. A summary of how the existing SPT and QOS

Profiles meet these requirements is also presented. The goal is to understand and clarify the premises for some of the requirements for quantitative annotations listed in the MARTE RFP, in order to refine them and to make sure that they are consistent, complete and capture all the expressive power needed for a future MARTE solution.

A requirement that is identified in the paper but is not specifically stated in the MARTE RFP is the need for global (i.e., model-wide) independent parameters for the evaluation, that capture application and platform quantities defining the parameter space for the evaluation. Such global parameters make the annotation more flexible and the analysis more robust. For instance, the quantitative annotations of an application can be given as functions (i.e., expressions) whose parameters are the platform characteristics. This allows for a better separation of modeling concerns and it's close to the whole MDA approach.

## References

[1] 4. V. Cortellessa, A. Pompei, "Towards a UML profile for QoS: a contribution in the reliability domain", In Proc. 4th Int. Workshop on Software and Performance WOSP'2004, pp.197 - 206, Redwood Shores, California, 2004.

[2] Object Management Group, "UML 2.0 Infrastructures Specification", OMG Final Adopted Specification ptc/03-09-15, 2003.

[3] Object Management Group, "UML 2.0 Superstructure Specification", OMG revised Final Adopted Specification ptc/04-10-02, 2004.

[4] Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification," version 1.0, formal/03-09-01, September 2003.

[5] Object Management Group, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QOS)", Adopted Specification, ptc/2004-06-01, June 2004.

[6] Object Management Group, "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), Request For Proposals, OMG document: ab/05-02-03, February 2005.

[7] D. C. Petriu and C. M. Woodside, "Performance Analysis with UML," in *UML for Real*, B. Selic, L. Lavagno, and G. Martin, Eds. Kluwer, 2003, pp. 221-240.

[8] M. Woodside, V. Vetland, M. Courtois, S. Bayarov, "Resource Functions for Performance Aspects of Software Components and Sub-Systems", pp 339-256 in "Performance Engineering", eds R. Dumke, C. Rautenstrauch, A. Schmeitendorf, A. Scholz, Lecture Notes in Computer Science no. 2047, Springer-Verlag, Mar. 2001.