Date: April 2008

UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification

Version 1.1
with change bars

OMG Document Number: formal/2008-04-08
Standard document URL: http://www.omg.org/spec/QFTP/1.1/PDF
Associated files*: http://www.omg.org/spec/QFTP/20060101 (Profile in UML2)
http://www.omg.org/spec/QFTP/20060102 (QoS Metamodel in UML2)

---

* original files: ptc/06-11-03 (Profile in UML2), ptc/06-11-04 (QoS Metamodel in UML2)

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT

LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , MOF™ , OMG Interface Definition Language (OMG IDL)™ , and OMG Systems Modeling Language (OMG SysML)™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org*

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Courier - 10 pt. Bold: Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm.*

UML Profile for Qos and FT Roadmap

Version 1.0: The source documents for this specification include:

Alpha: ptc/2005 (submission)
Associated Schema files: ptc/2005-06-09 (xml), ptc/2005-06-10, ptc/2005-06-11 (metamodel)

Version 1.1: The source documents for this specification include:

Formal version 1.0: formal/2006-05-02,

Convenience: ptc/2006-11-05

Associated Schema files:

1 Scope

This specification defines a set of UML extensions to represent Quality of Service and Fault-Tolerance concepts. These extensions reduce the problems of UML 2.0 for the description of Quality of Service and Fault-Tolerance properties, and integrate the extensions in two basic general frameworks (QoS Modeling Framework, and FT Modeling Framework).

The general framework for the description of QoS requirements and properties gives the support to describe vocabulary that we use in high quality technologies (e.g., real-time, fault-tolerant). The framework provides the ability to associate the requirements and properties to model elements to introduce extra-functional aspects in UML models. High quality systems must allocate their extra-functional requirements in the analysis models, and must support the decomposition of these requirements in the software architectures.

The general framework for fault-tolerance includes notations to model risk assessments, paying special attention to the description of hazards, risks, and risk treatments. This general framework supports the description of fault-tolerant architectures based on object replications.

2 Conformance

Compliance with this specification is defined as compliance with any of the following profile packages and model libraries:

- QoS subprofile package: Profile models of Chapter 9. This chapter includes in Section 9.4 a subprofile package that extends the GRM profile package of UML Profile for Scheduling, Performance, and Time standard.
- QoS Catalog model library. Model library of Chapter 10.
- Risk Assessment subprofile package: Profile models in Section 11.2 and graphical notations in Section 11.4.
- FT subprofile package: Profile models in Section 12.2.

Compliance with a profile package implies complying with any packages on which the particular package depends via a package dependency, complying with the profile models, and with the profile constraints. Compliance with a model library implies complying profile packages prerequisites for the model library and other model libraries on which the particular package depends via a package dependency, and the constraints included in the model library.

3 Normative References

The normative references of this standard are:

- Object Management Group, UML 2.0 Superstructure Specification, OMG document number formal/05-07-04.
- Object Management Group, UML Profile for Scheduling, Performance, and Time Specification, OMG document number formal/2005-01-02.

4 Terms and Definitions

There are no formal definitions in this specification that are taken from other documents.

5 Symbols

There are no symbols defined in this specification.

6 Additional Information

6.1 How to Read this Specification

This specification includes two different parts. The first provides solutions to the mandatory requirements related to QoS (Quality of Service) specifications. The second are solutions to the requirements for the FT (Fault Tolerance) aspects. The table below includes the contents of the following sections.

Chapter 7	Introduces the objectives of this specification and its general structure.
Chapter 8	This chapter includes the metamodel of QoS Framework that the RFP request as mandatory part of the specification.
Chapter 9	This chapter includes the UML profile of QoS Framework.
Chapter 10	This chapter includes a proposal of QoS Catalog that introduces a set of domain and application independent QoS characteristics, which are supported with the QoS Catalog.
Chapter 11	This is the first chapter of the second part of this specification. It includes methods for the description of models of risk analysis.
Chapter 12	This chapter includes notations for the extension of UML for the description of fault tolerant architectures.
Annex A	This annex concretes the conceptual elements that include the standard *UML Profile for Scheduling, Performance, and Time*, OMG document number ptc/2002-11-01 (November 2002). for the description of QoS of resources. QoS Framework and profile reuses these concepts for the description of QoS of resources.
Annex B	This annex presents the application of QoS Framework for the description of latency properties that include the standard *UML Profile for Scheduling, Performance, and Time*, OMG document number ptc/2002-11-01 (November 2002). And uses these characteristics for the generation of scheduling analysis models.
Annex C	References

6.2 Acknowledgements

The following companies submitted and/or supported parts of this specification.

- **Ilogix-Inc.**
- **Open-IT**
- **THALES**
- **ARTISAN**
- **CEA**
- **Lockheed Martin**
- **SINTEF**
- **Softeam**
- **Universidad Politecnica de Madrid**

The following persons designed and wrote this specification: Jan Øyvind Aagedal, Miguel A. de Miguel, Emmanuel Fafournoux, Mass S. Lund, and Ketil Stølen.

In addition, the following persons contributed valuable ideas and feedback that significantly improved the content and the quality of this specification: Arne J. Berre, Earl F. Ecklund, Sebastien Gerard, Eric Jouenne, Benoit Langlois, Alan Moore, Laurent Rioux, Bran Selic, Chris Sluman, Bennett C. Watson.

7 Rationale and General Models

Frequently the behavior of a system component is functionally correct, but the result it generates is nevertheless unacceptable because the result does not meet some QoS criteria, such as the response time and accuracy (i.e., quality). One way to enhance the capability of the system to deliver results of acceptable quality is to use flexible components. A *flexible* component can trade off among the amounts of time and resources it uses to produce its results, the quality of its input, and the quality of its result.

In addition to its functional behavior and internal structure, the developer of each component must consider its QoS requirements. For example, components such as pattern recognizers or signal filters have temporal requirements (e.g., maximum response times and jitters and minimum execution frequencies) and input and output accuracy requirements (e.g., percent of error in the pattern recognition as a function of noise in the input). If the component is flexible, the output quality depends both on input quality and available resources (e.g., amounts of CPU execution time and memory).

Most modeling languages provide support for the description of functional behavior, they describe non-functional requirements merely using simple comments or informal structures. An example are the interfaces that provide support for the description of functional services in some modeling and interface description languages, but they do not specify non-functional properties of implementators. When a client defines a dependency of these interfaces, it has no information about the quality properties.

QoS can be defined as a set of perceivable characteristics expressed in user-friendly language with quantifiable parameters that may be subjective or objective [25]. Examples of objective parameters are startup delay and data sizes. Subjective factors are the overall cost or the factors of importance of other parameters. Examples of QoS parameters for system resources are jitters, delays, blocking times, and size of buffers.

The characteristics of quality and their parameters are based on two types of concerns: i) user satisfaction, parameters are based on the user or client requirements, and ii) resource consumption and system parameters, these are the parameters that support the resource managers of system infrastructures. Sometimes the user parameters depend on some properties of the functional architecture (e.g., types of algorithms, data redundancy, and limited execution time). In the process of analysis of QoS, we must establish the mapping between different user parameters and the resource parameters or the functional architectures to achieve the user qualities based on system parameters and the functional implementation of the system.

The function $f(qi, r) \rightarrow qo$ do the quality characterization of software components or the entire system, where *qi* are the quality attributes of other components or external environment that affect the quality of this component (input), *r* are the resources used in the component that affect to its qualities, and *qo* are the qualities provided. Examples of input and output qualities are the precision of input/output arguments, maximum frequency of input/output data, and the accuracy of output results (output). Examples of resource qualities are the maximum response time in CPU executions and network bandwidth. The function depends on the functional behavior (e.g., to support reliability we must include some type of redundancy in the architecture or implementation).

7.1 Constructors of QoS Modeling Languages

QoS specification languages are based on a set of constructors that provide support to describe the main QoS elements of the problem. Nevertheless, the model requires a general reference architecture. We are going to consider the QoS specification for two different abstraction levels: QoS application analysis and QoS application architecture. In the first case we analyze the QoS of the system that is going to be developed and in the second case we study the QoS of solutions. The general model for the QoS application architecture is based on ISO general QoS architecture [12].

The basic functional elements of the QoS model that we will be considering is the resource-consuming component (RCC) for the QoS application architectures and the QoS-aware specification functions (QASF) for the QoS application analysis. QASF are significant services and functions of the new systems (specified from an analysis point of view) that have associated QoS requirements. These functions support the functional behavior of the system. However, the execution of each function will take time, require system resources, and be subject to occasional system errors or failure. These and other similar features are non-functional behavior of the system. RCC is a processing entity that includes a group of concurrent units of execution, which cooperates in the execution of a certain activity and share common budgets. The budget is an assigned and guaranteed share of certain resources. An RCC has the following associated: i) facets (interfaces provided and synchronously used by RCC clients), ii) receptacles (interfaces synchronously used by this RCC), iii) event sinks (event queues supported by this RCC and asynchronously used by RCC clients), and iv) event sources (event queues asynchronously used by this RCC). UML can model RCC in different ways; in general, classes, components, and interfaces are modeling elements that model the RCCs. At this point what we want to address is the identification of the main concepts that a QoS model includes.

QASFs and RCCs have non-functional characteristics associated, which can be general purpose or domain specific. In both cases QoS characteristics make reference to quantifiable non-functional attributes. The quantification with one or multiple dimensions is fundamental for the expression of QoS supported-provided, the monitoring of the characteristic, and evaluation of fulfillment and level of satisfaction.

A **quality characteristic** includes a set of quality attributes that are the dimensions to express a quality satisfaction. An example of quality characteristic to express latency constraints could include the following attributes: i) arrival patterns, the values of this enumerated quality value are: periodic, irregular, bounded, busty, unbounded, ii) minimum period, iii) maximum period, iv) jitter, v) burst interval, vi) burst size, vii) requirement type, the values of this enumerated quality value are: hard, soft, firm, viii) deadline hard, ix) deadline soft, and x) output jitter.

The facets, receptacles, event sinks, and event sources interconnect the RCC group, which collaborate to provide support of QASF. They support QASF transforming input data and events into output data and events. The QASF are the external QoS system operations, which have a quality utility associated that express the degree of satisfaction of the operation, from the user or external system point of view. The quality utility is expressed in terms of quality types and quality constraints. The grouped RCC are not quality independent in the sense that their configuration and quality provided in their facets and event sink may limit the quality behavior of another RCC. The end-to-end quality of a qualified functionality depends on the sequence of transformations developed along the RCC sequence. For example, the end-to-end latency of a video signal transformation depends on the latency of all RCCs involved in the transformation operation.

Quality levels express the quantifiable level of satisfaction of a non-functional property. An RCC can associate quality levels to its facets and event sinks. These quality levels are the RCC quality provided contracts. To support the quality provided contracts, the RCC can require some minimum budgets and quality levels in its receptacles and event sources, and in the system resources. These quality levels are expressed in the **required quality contracts**. Quality contracts are expressed in terms of the values associated to quality characteristics.

7.2 QoS Modeling Elements

A general QoS modeling language must provide support for the specification of:

- *Definition of QoS Characteristics*: *QoS Characteristic* is a quantifiable aspect of QoS, which is defined independently of the means by which it is represented or controlled [12]. *QoS Characteristics* are quantified with some specific parameters and methods, and with other characteristics with a lower abstraction level. QoS Characteristics can be grouped into categories that group characteristics of a common subject. Different enterprises and organizations use the same QoS Characteristics, but they use different evaluation methods or establish different hierarchies. An example of divergence is that standards like [13] do not identify specific QoS characteristics for performance, but other proposals like [4] do. Another specific example is what we can do to measure *availability* characteristics. We can do it with

different levels of abstraction: i) a simple probability, and in a hypothetical domain, this is enough, and we do not enter in more details, ii) in other domains we require more details, and we use two arguments, the mean-time-to-repair (MTTR) and the mean-time-to-failure (MTTF), and the *availability* is the probability: MTTF / (MTTF + MTTR). iii) When the operations are transactions, this probability is not enough, and we need to introduce the availability period (the period that a client will be able to access times arbitrarily) and the availability makes reference to a *continuous availability* [4][15]. Some authors would classify the last case such as a *reliability* quality. Different domains require different levels of abstraction. We need enough flexibility to make the description of particular characteristics of specific domain environments possible.

- *QoSConstraint*: The *QoS Constraints* define any kind of restriction that QASF and RCC impose on *QoS characteristics*. The restrictions express limitations in the parameters and methods of characteristics. They identify ranges of values allowed for one or multiple parameters and methods and their dependencies. Examples of simple *QoS Constraints* are constraints that describe maximum response times, or the minimum number of errors supported. Sometimes the *QoS Characteristics* have associated interdependencies, for example, in a compression algorithm; the response time depends on the compression degree (more level of compressions, requires more computation time) or the functions for the description of subjective priority of qualities or for the description of quality optimal values. Figure 7.1 represents the dependencies of qualities qx, qy, and qz for a hypothetical implementation function. Figure 7.1 represents the maximum and minimum values and the dependencies of quality values; qx cannot have an arbitrary value when the values of qy and qz are fixed. Analytical methods are based on the optimization of these functions and these functions can be restricted for specific analysis methods.

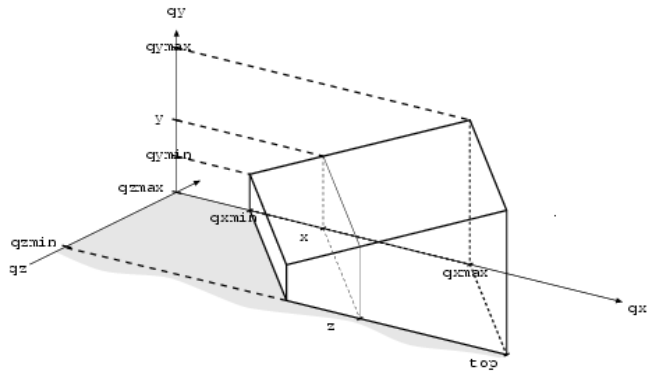

Figure 7.1 - Relationship of qualities

- *QoS Levels of Execution*: Sometimes, QASFs and RCCs are designed to support different modes of executions with different quality levels. Each mode has different *QoS Constraints* associated and their functionality can be different. Often, the execution modes for specific QASF or RCC are discrete (there is an enumeration of the different modes), for example, the levels of quality of a window in a digital television are High, Medium, and Low resolution, and the television system uses different implementation algorithms for each mode [21]. But in some cases, the mode is defined with continuous values, for example, the maximum speed of the target in radar, and the quality level supported is specified with the values of the speed. In this case, the level is based on the real number that describes the speed. In general, the design of the functional architecture must take into account these modes, and there are different functions and components for each mode.

- *QoS Adaptation and Monitoring*: The transition from one execution mode to another requires some actions in the application execution, and some types of transitions are not allowed (we cannot change the quality level arbitrarily). Another common activity in some applications is monitoring QoS characteristics for the detection of errors and robustness. The monitors detect non-achievement of some QoS constraints, but we must specify the actions to be taken when the system does not achieve the quality levels.

Specific reservation protocols, admission control and analysis methods must specialize these general elements and restrict allowed values. They address solutions for platform independent models. Specific QoS frameworks that provide specific supports to the negotiation process require the specialization of some elements.

8 QoS Framework Metamodel

This metamodel defines the abstract language of a modeling language that supports modeling general QoS concepts. This metamodel defines a set of modeling elements represented as metaclasses. A concrete syntax must define the specific notation rules for the graphical representation of this modeling language. In our case the concrete syntax does not exist, and the UML profile that we will introduce in the next section supports the representations of QoS concepts in models (in UML models).

Because we intend to use UML notations for the representation of QoS concepts, some elements in the abstract syntax have been simplified to reduce its complexity. For example, in Figure 8.2 association `Template-Derivations` defines a relationship between *QoS Characteristics*. There is no metaclass to represent the model element that represents the relationship.

8.1 General QoS Framework

A general QoS framework provides support to ensure consistency in modeling various qualities. The QoS framework supports a general categorization of different kinds of QoS; including QoS that are fixed at design time as well as ones that are managed dynamically. Futhermore it supports the integration of different categories of QoS for the purpose of modeling QoS of system aspects. This section includes the metamodels that describe the main packages of this UML extension. The different metamodels establish the elements used to model QoS systems.

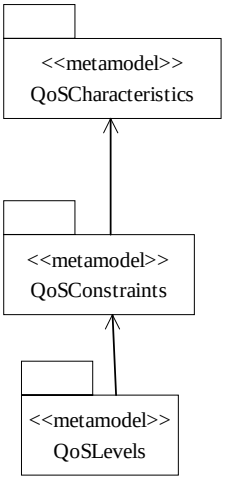

Figure 8.1 - Submetamodels in the QoS Metamode

Figure 8.1 includes the set of packages that comprise the metamodel. The QoSCharacteristics package includes the model elements for the description of QoS Characteristics; the QoS Constraints package includes the modeling elements for the description of QoS contracts and constraints and the QoS Levels package includes the modeling elements for the specification of QoS modes and transitions. The next section specifies each of these metamodels.

QoS metamodels can be used to create repositories for handling QoS metadata in different kinds of tools. The metamodels that we include in this chapter include references to UML2 metamodels (merged versions of UML2 metamodel), and they can be integrated with UML2 modeling frameworks. But, if we remove these references, they have been designed to be reusable in other frameworks or modeling languages.

8.2 QoS Characteristic

Figure 8.2 includes the metamodel for the description of QoS Characteristics. The concepts involved are listed below.

QoS Characteristics represents quantifiable characteristics of services. The *QoS Characteristics* are specified independently of the elements that they qualify. *QoS Characteristic* is the constructor for the description of non-functional aspects like: latency, throughput, capacity, scalability, availability, reliability, safety, confidentiality, integrity, error probability, accuracy, and loading as mentioned in [4][12][13][15]. These are some general characteristics, but specific domains have associated specific characteristics.

The relation `qSub-qParent` provides support for the extensions-specialization. This association supports the reuse of characteristics. Sometimes the *QoS Characteristic* definition requires some parameters. The description of generic characteristics may require, for example, the parameterization of the units and types for the description of value definitions, or some specific methods for the quantification of the values; *QoS Parameter* supports these parameters. The attribute *isInvariant* specifies when the QoS Characteristic can or cannot update the value of dimensions dynamically.

QoS Dimension: QoS Dimensions are dimensions for the quantification of *QoS Characteristics*. We can quantify a *QoS Characteristic* in different ways (e.g., absolute values, maximum and minimum values, statistical values). For example, we can quantify the latency of a system function as the end-to-end delay of that function, the mean time of all executions, or the variance of time delay. A *QoS Characteristic* can require more than one type of value for its quantification. Examples of dimensions of *Reliability* are (from [9][15]): time to repair; time to failure; failure masking that server exposes to their clients (failure, omission, response, value, timing, late or early); service failure is the way in which a service can fail (halt, roll back, or initial state); semantic of services (exactly once, at least once, or at most once); and number of failures supported. We can define a dimension of a *QoS Characteristic* based on another *QoS Characteristic* and we make composition of QoS Characteristics for the definition of new qualities. The type of dimension specifies the quantification.

To make a relational comparison between two values of *QoS Dimensions*, which domain is ordered, we need to know the relational precedence of the domain. The attribute *direction* (an enumeration of *increasing*, *decreasing*, and *undefined* values) defines the type of order relation. When the attribute is *increasing* the relation ´>´represents a *higher-quality* relation, when it is decreasing, it is the relation ´<´. When the relation is *decreasing*, the quality value *x* is better than *y* when $x < y$ is true. For example, in most of the systems, a low *response time* is better than a high response time; in this case, the value is *decreasing*. The *rate transmission* is an example of *increasing* value, because, in general, a high transmission rate is better. *Units* allows the specification of the unit for the values of the dimension and *statisticalQualifier* gives the type of statistical qualifier when the value of the dimension represents a statistical value. The types of statistical qualifiers are: *maximum value*, *minimum value*, *range*, *mean*, *variance*, *standard deviation*, *percentile*, *frequency*, *moment*, and *distribution*.

QoS Category: When the number of *QoS Characteristics* is large, or they are especially complex, some mechanisms for grouping are required. Some examples of general groupings of quality attributes are: i) *Performance*: Performance makes reference to the timeliness aspects of how software systems behave. ii) *Dependability*: Dependability is the property of computer systems such that reliance can justifiably be placed on the service it delivers. iii) *Security*: this capability covers different subjects such as the protection of entities, and access to resources.

The main difference between a *QoS Category* and a *QoS Characteristic* is that *QoS Characteristics* are directly quantifiable, and *QoS Categories* do not provide a direct framework for the evaluation of non-functional attributes; it requires a more detailed level of specification to establish constraints or comparisons. The definition of *QoS Characteristics* included in a *QoS Category* can make reference to characteristics defined in other categories. The property *referenced* represents this dependency.

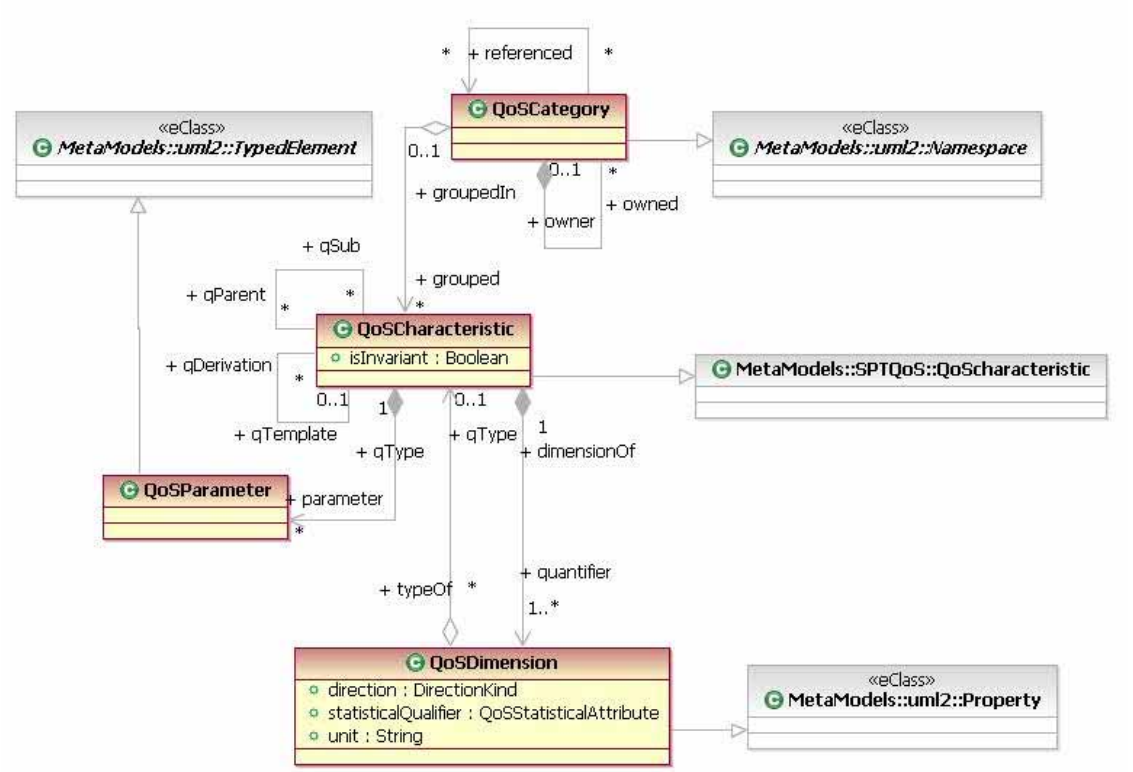

Figure 8.2 - QoSCharacteristic Diagram

In general, *QoS Characteristics* are defined independently of the modeling element that they describe and the other *QoS Characteristics*.

Figure 8.3 includes the metamodel for the description of *QoS Values*. The concepts involved are listed below.

QoS Value: QoS Characteristics and *QoS Context* provide support for the description of quantifiable QoS values. However, often there are some QoS specific values identifiable at modeling time (e.g., limits of characteristics, or specific QoS values). *QoS Value* instantiates *QoS Characteristic* and fixes it with specific values of its value definitions (*QoS DimensionSlot*). When we attach a *QoS Value* to a model element, we are characterizing the element with quality values.

QoS Value and QoS Characteristics are specializations of `QoScharacteristics` and `QoSValue` metaclasses included in the package `Core Resource Model` of standard *UML Profile for Scheduling, Performance and Time* (SPT) [20]. These specializations allow reuse of these concepts modeling QoS for resources, see Figure 8.3.

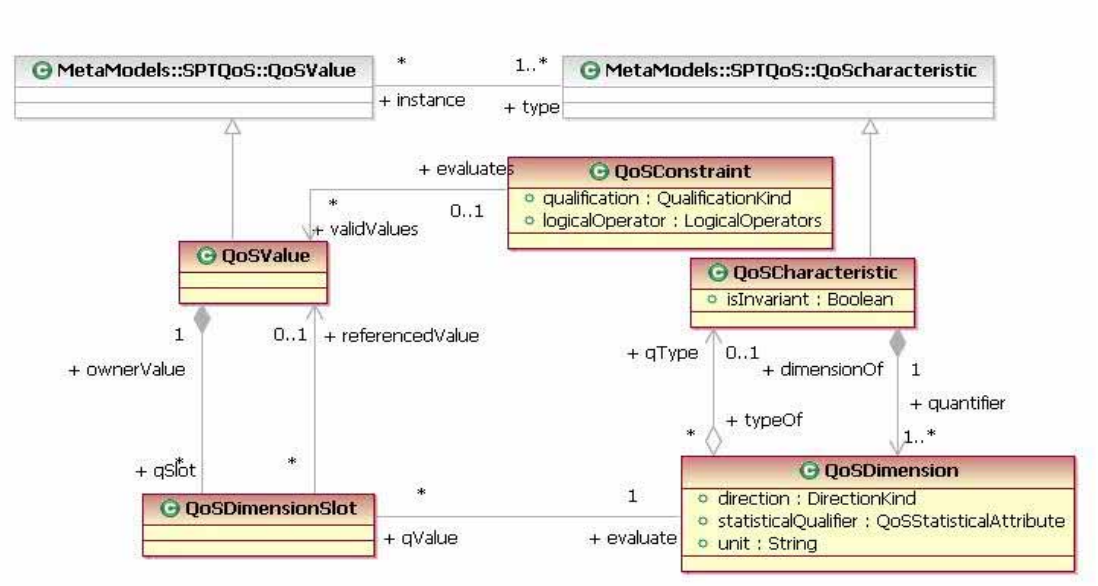

Figure 8.3 - QoSValues Diagram

QoS Values are instances of *QoS Characteristics* that have resolved all their parameters. The OCL expression that expresses this is:

```
context QoSValue inv:
    self.type.parameter->size() = 0
```

QoS Dimension Slot: *QoS Dimension Slot* represents the value of a primitive *QoS Dimension* or a reference to another *QoS Value*. The attribute *Value* has valid value when the slot represents primitive dimensions. The association `referencedValue` identifies the value that references a non-primitive dimension.

Figure 8.4 includes the metamodel for the description of QoS Context. The concepts involved are listed below.

QoS Context: Often, quality constraints and expressions have more than one *QoS Characteristic* associated. Sometimes, these expressions and constraints combine functional elements and non-functional elements. *QoS Context* allows describing the context of quality expression when it includes multiple *QoS Characteristics* and model elements. A single *QoS Characteristic* defines a *QoS Context* for the expression whose references are only to the *QoS Characteristic*. We can define a *QoS Context* based on other *QoS Contexts* and *QoS Characteristics*.

The attribute *isQoSObservation*, whose default value is *false*, defines when a *QoS Context* represents an environment of quality observation. A quality observation records quality values of *QoS Characteristics* included in the association *BasedOn*. It records the *QoS Characteristics* of *QoS Context* included in this *QoS Context*, if they are not observation contexts. A quality observation is associated to a single execution element (e.g., a link end, a message, an object), and records values of *QoS Characteristics*. This allows representing constraints that include more than one quality point (e.g., end-to-end quality constraints, or output qualities that depend on input qualities).

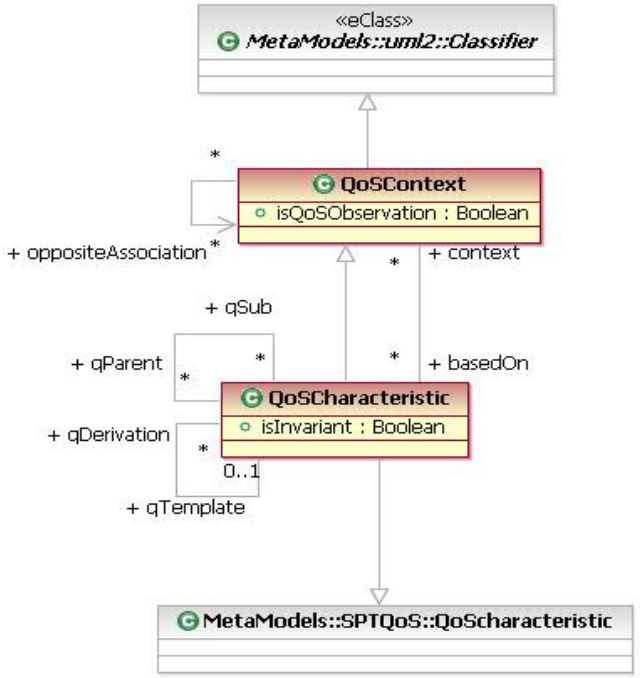

Figure 8.4 - QoSContext Diagram

A *QoS Context* is described with *QoS Characteristics* or with the combination of other *QoS Contexts*. This means that a *QoS Context* that does not include another *QoS Context* must be defined in terms of *QoS Characteristics*:

```
context QoSContext inv:
        self.oppositeAssociation.size() = 0 implies self.basedOn.size() > 0
```

8.3 QoS Constraint

Figure 8.5 includes the metamodel for the description of QoS Constraints. In this diagram we identify one abstract metaclass (QoS Constraint) and three specific types of QoS contracts. The concepts involved in the constraints description are listed below.

QoS Constraint: This is an abstract metaclass. A *QoS Constraint* limits the allowed values of one or more *QoS Characteristics*. The *QoS Constraints* define the constraints of the *QoS Characteristics* of modeling elements. Application requirements or architectural decisions limit the allowed values of quality and the *QoS Constraints* describe these limitations. *QoS Context* defines the *QoS Characteristics* and functional elements involved in a *QoS Constraint*. The *QoS Context* establishes the vocabulary of the constraint, and the *QoS Constraint* allowed values. Two approaches for the description of values are:

1. Enumerate the *QoS Values* allowed for each of the *QoS Characteristics* involved in the *QoS Context*. For example, we can specify the allowed latency and accuracy for a function that computes the trajectory of a target in military radar. In this case, both *QoS Characteristics* are considered independent.

2. The expressions that must be fulfilled by the *QoS Characteristics*. These expressions define maximum and minimum values, and the dependencies of *QoS Characteristics*. This approach is more flexible than previous ones, because we can identify not only the limits but also the dependencies of both *QoS Characteristics* supported. We can identify

some kind of relationship between accuracy and latency (for a more precise trajectory we require more computation time). Furthermore, the allowed values are not independent.

We can see the quality constraints from two points of view: from the client's point of view and from the provider's point of view. This approach defines two different types of constraints: constraints required and constraints offered (Figure 8.5). This is a common approach in the specification of QoS [1][2][6][23]. Some approaches pay special attention to the quality provided by the resources and the impact of sharing the resources, and others pay more attention to the extension of functional interfaces with non-functional contracts.

QoS Required: when a client defines its *Required QoS* constraint, the provider (software element or a resource) that supports the service must provide services with some quality levels to achieve its clients' requirements. The service provider must support not only the service required but also must provide its services with certain quality constraints. When the provider defines its *Required QoS* constraint, the client must achieve some quality requirements to get the quality offered. An example of *Required QoS* constraint for providers is the maximum frequency of invocation from its client.

When a client uses a service with some quality requirements, it specifies the non-functional requirements with a *Required QoS* constraint. This constraint specifies the quality that the server must achieve. This constraint limits the space of valid values for the *QoS Characteristics* involved in the service. The *QoS Characteristics* are the dimensions of the quality space, and the *QoS Required* defines the valid values of this space.

The expressions of *Required QoS* constraints appear because the system must fulfill some user requirements, or because a user-provider (a software element that provides services and uses other services) component or subsystem must support other qualities required. The user-provider requires some qualities for its providers to achieve the quality required. For example, a reliable component can support its quality requirements, when its service providers are reliable.

Often, an end-to-end quality requirement is decomposed into a set of sub-quality requirements, and the software architects define a set of *Required QoS* constraints to achieve the quality required.

QoS Offered: The specification of software components includes the description of their interfaces. The interfaces include the set of services provided. However, their architectures and implementations are designed to support some specific qualities. Architects design a component or a subsystem to support some levels of scalability, to have a limited response time, or to make the component reliable based on persistence techniques, and these decisions create impacts in the types of data structures and algorithms. These quality properties have a special impact on the architecture.

QoS Offered has associated the set of *QoS Characteristics* that the component takes into account (*QoS Characteristics* are part of the specification of *QoS Context*). *QoS Offered* establishes the limits of values that support the software elements. This is the space of quality that can support the software element. When a quality does not appear in the context of the *Offered QoS*, the software element does not take it into account, and the component does not guarantee this quality. Often, the *Offered QoS* depends on the QoS provided by the resources and service providers that the software element uses. When the provider defines an *Offered QoS* constraint, it is the provider who must achieve the constraint. When a client defines an *Offered QoS* constraint, the client must achieve the constraint invocating the service.

QoS Contract: The quality provider specifies the quality values it can support (provider- *Offered QoS*) and the requirements that must achieve its clients (provider-*Required QoS*). And the user specifies the quality it requires (client-*Required QoS*), and the quality that it ensures (client-*Offered QoS*). Finally, in an assembly process, we must establish an agreement between all constraints.

In general, the allowed values that client-*Required QoS* specifies must be a subset of values supported in provider-*Offered QoS*, and the allowed values that provider-*QoS Required* specifies must be a subset of values supported in client-*Offered QoS*. If the provider does not support the QoS required, we must *negotiate* the contract and the final quality provided. Sometimes, we cannot compute the *Contract QoS* statically, because it depends on the resources available or quality

attributes fixed dynamically. However, we can identify the qualities supported in the contract and some limit values. To compare two *QoS Values*, we can use the = operator for *QoS Dimension Slot*, or we can use the attribute *direction* in *QoS Dimension*. In the last case the data type of *QoS Dimension* associated with *QoS Dimension Slot* must support the relational operators < and >.

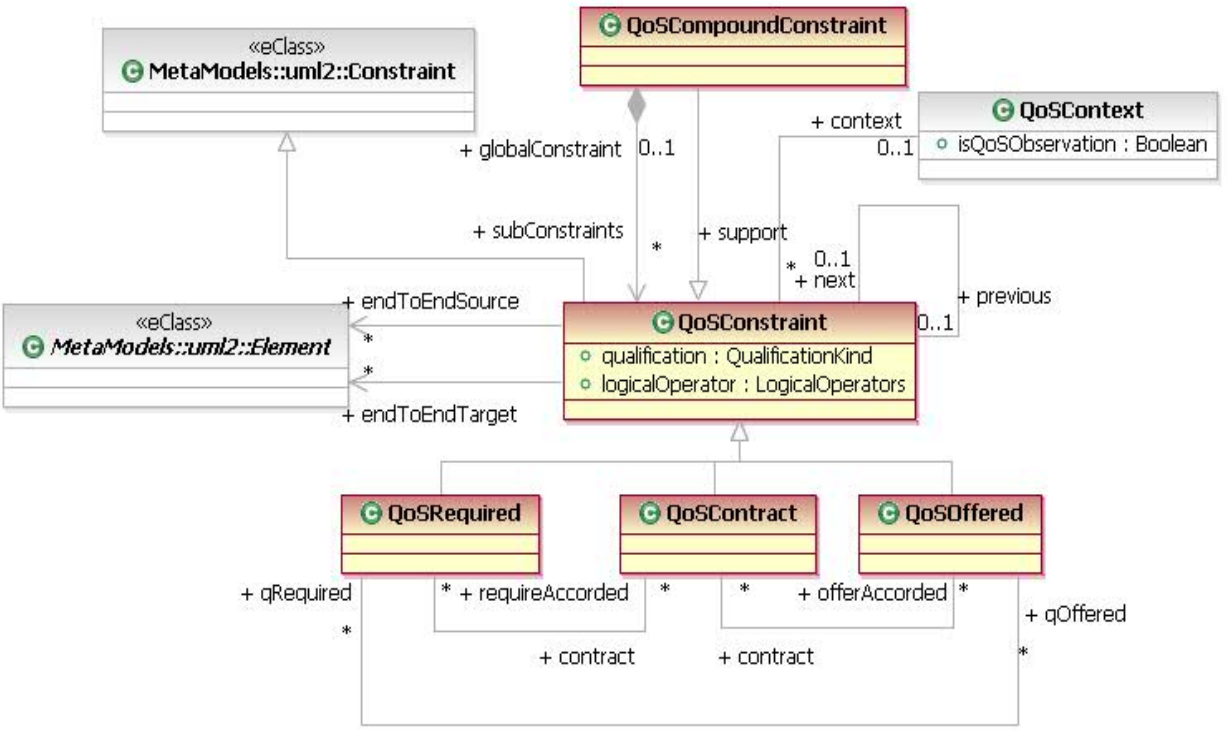

Figure 8.5 - QoSConstraint Diagram

Some infrastructures provide support for the management of *QoS Contracts*. A common approach is to extend the IDLs (Interfaces Description Languages) [5][27] to provide support for the description of quality contracts. QuO [27] is an example of an environment for the specification of QoS contracts and management of negotiation processes, and access to *Resource Managers* and services to support some types of quality such as reliability. Other approaches are based on component solutions [7][26], and the component infrastructures support the agreement and adaptation to the quality contracts.

The attribute *qualification* specifies the strictness of the constraint. The values are: *Guarantee*, *Best-Effort*, *Threshold-Best-Effort*, *Compulsory-Best-Effort*, and *none*.

End-to-End QoS constraints make reference to quality constraints that involve two or more modeling elements. Source elements define source instants for the quantification of constraints and target define the end of interval. *endToEndSource* and *endToEndTarget* are properties that make reference to source and target modeling elements. An example would be a QoS constraint (for example an OCL constraint attached to two modeling elements) that makes reference to two actions in an activity diagram (an iteration diagram would be the similar). Sometimes we can identify implicitly who are the source and the target, but in some cases (for example loops) this implicit assumption can generate errors. *endToEndSource* and *endToEndTarget* properties define explicitly the sources and targets.

Often a model element does not have a single mode of execution or can adapt its execution to provide different quality levels. This means that the quality offered or required can be a combination of a set of quality constraints, and each constraint is associated to a specific level or execution mode. *QoSCompoundConstraint* is the combination of a set of constraints that ensemble represent a QoS Constraint for the model element (e.g., class, component, or object).

QoSCompoundConstraints can represent global constraints decomposed in a set of subconstraints. The sub constraint can have a precedence order relations. These relations can represent, for example, how to decompose a latency constraint in a set of subconstraints.

A *QoS Constraint* must have associated at least one *QoS Context* that is the reference for the description of expressions and values for the software element associated to the constraint. However sometimes the QoS value for a software element depends on the allowed QoS values of other software elements. For example, the QoS that a component offers in its interfaces depends on the QoS values required in the used interfaces. A *QoS Constraint* can reference multiple contexts and the context can identify software element that is the source of dynamic QoS values.

The *QoS Compound Constraint* qualification is defined in terms of the *QoS Constraints* that it includes, and because of this the value of attribute *Qualification* is the default value (none):

```
context QoSCompundConstraint inv:
   self.qualification = QualificationKind::none
```

8.4 QoS Level

Figure 8.6 is a metamodel for the description of *QoS Levels*. The basic concepts of this model are listed below.

QoS Level: QoS Level represents the different modes of QoS that a subsystem can support. Depending on the algorithms or the configurations of the systems, the component can support different working modes, and these working modes provide different qualities for the same services. For each working mode, we specify a *QoS Level*. The *QoS Levels* represent states in the system from a quality point of view. The current *QoS Level* depends on the current resources available, the quality required, and functional parameters such as state variables that identify the current configuration. For each *QoS Level* the resources required are different. In general, the resources offer different quality depending on the load that they have. *allowed Space* describes the conditions that a software element and the system must achieve to state in a specific *QoS Level*.

When a *QoS Level* has more than one *Allowed Space*, the system continues in the *QoS Level* if all *allowed Space* expressions are true.

A *QoS Level Change* occurs when the *allowed Space* of the current *QoS Level* becomes false, and a transition fires. This change must have one enabled transition from the current *QoS Level* to another that is going to be fired. If there is not an enabled transition, the system is in a state where it cannot achieve its QoS requirements, it will continue in the current state, but it cannot support its contracts.

An example of change is when the resources cannot support their contracts (they have received new requests and the resource has a different load level), and we must change the *QoS Level* of some elements. If we cannot change the level, the component will not fulfill its QoS requirements. This change initiates a process of adaptation in the quality of the component. Another source of *QoS Level Change* is the reconfiguration of the component. The component must provide different levels of quality because the user requires a different level of quality at the components with external interfaces. Examples of these changes occur in multimedia systems, when the user resizes some video windows or changes the quality levels of audio and video.

QoS Transition: QoS Transition models the allowed transitions between *QoS Levels*. *QoS Level Change* is a type of event that can fire a *QoS Transition*. The architecture and implementation of software elements take into account these states and transitions. The property *adaptionActions* includes the set of actions to describe the adaptation to new mode.

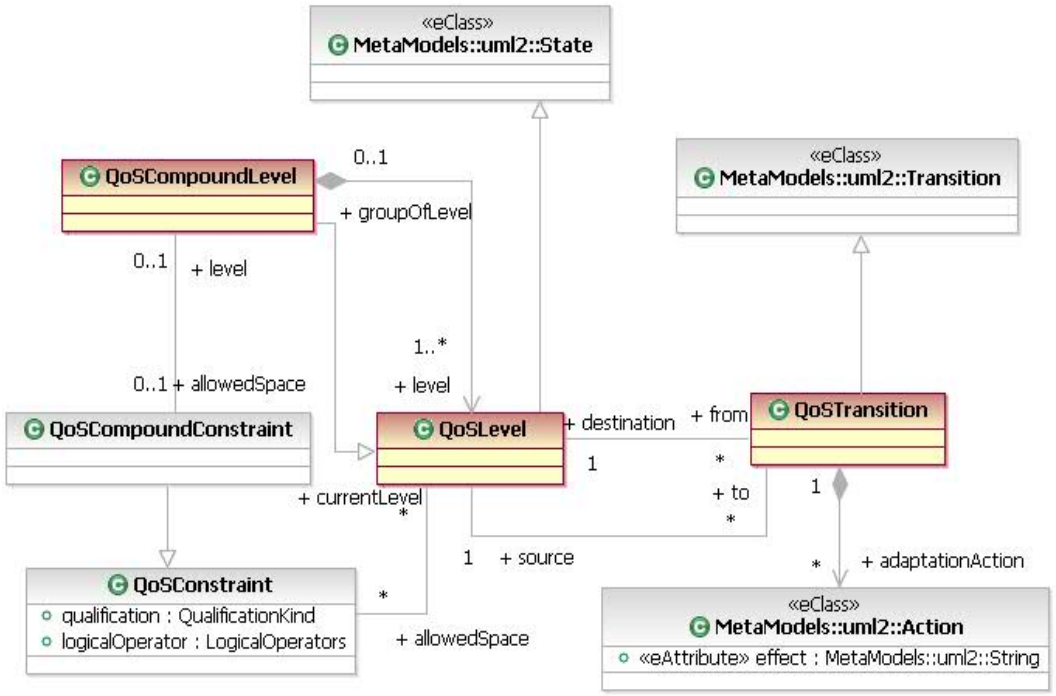

Figure 8.6 - QoSLevel Diagram

The *QoS Levels* define the QoS behavior of model elements. This behavior must fulfill that when the quality state is some quality level, the *QoS Constraint* associated to it must be fulfilled.

```
context QoSLevel inv:
     self.allowedSpace->forAll(self == true)
```

A QoS Compound Level includes all the *QoS Levels* that define the quality behavior of a model element. These *QoS Levels* have associated some *QoS Constraints*, all of them define the QoS Compound Constraint associated to the QoS Compound Level:

```
context QoSCompoundLevel inv:
     let constraintsOfQoSLevels : Set(QoSConstraint) = self.allowedSpaces in
            self.level->forAll(constraintsOfQoSLevels->
                 includesAll(self.allowedSpaces))
```

8.5 Integration with Package Core Resource Model

The modeling elements that we have introduced provide support for the description of *QoS Characteristics* and contracts in general. One specific case is the quality provided by resources and their contracts. SPT [20] pays special attention to this type of qualities, and includes classes that concrete the relations between resources and qualities.

Three packages of SPT can support the description of QoS contracts for resources: `GeneralResourceModel::CoreResourceModel`, `GeneralResourceModel::ResourceUsageModel`, and `GeneralResourceModel::CausalityModel`. CoreReourceModel includes the classes *QoScharacteristic* and *QoSValue* that we reuse in the metamodel of package QoSCharacteristic.

Annex A includes the SPT diagrams that we reuse to support the description of QoS for resources. *General Resource Model* in SPT was designed as a conceptual model and not as a metamodel. It was not designed to identify modeling structures and because of this some concepts included in the diagrams in Annex A cannot be identified in annotated UML model but others can be identified with profile annotations (e.g., resources, resource usage, and client).

9 UML QoS Profile

This profile defines *limited extensions* to the reference UML 2.0 meta-model with the purpose of adapting the meta-model to a specific platform or domain. The extension of this profile does not change the reference metamodel, and keeps its semantics.

In UML 2.0, profiles are packages that structure UML extensions. The principal extension mechanism in UML 2.0 is the concept of *stereotype*. Stereotypes are specific metaclasses, having restrictions and the specific *extension* mechanism. Additional semantics can be specified using Stereotype features (“attributes” in UML2.0, “tagged values” in UML1.x) and new well-formedness rules in the context of a profile.

A UML profile extends parts of the UML metamodel in a constrained way. All new modeling concepts must be supported by UML modeling elements. The new attributes must respect the semantic of UML modeling elements. All associations are binary associations. We cannot redefine features, but we can add new features (meta-attributes of stereotypes). UML metaclasses are extended by stereotypes, using a mechanism called *extension*. The semantic of metaclass generalization and stereotype extension must not be confused. We will use this notation of UML 2.0 to represent this profile.

The general structure of the profile model is the same as the general structure of the metamodel.

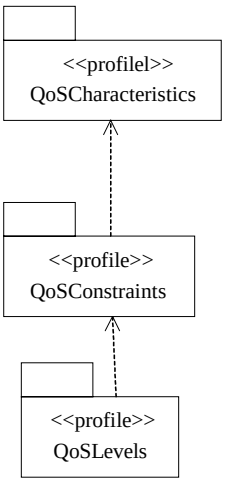

Figure 9.1 - Subprofiles in the QoS Profile

9.1 QoS Characteristics Subprofile

The base class of stereotype `<<QoSCharacteristic>>` is, in general, *Class*. *Properties* and *Structural Features* with stereotype `<<QoSDimension>>` provide support for the quantification of characteristics. The base classes of `<<QoSDimension>>` are *StructuralFeature* and *Property.* The types for *QoSDimensions* are UML 2.0 primitive types, enumerations, or *QoS Characteristics*. The metaclass *Class* included in metamodel *Classes::Kernel* does not provide support for the description of template properties. The metamodel *AuxiliaryConstructs::Templates* includes the constructors for the description of template and parameters of model elements. *Classifier* metaclass in metamodel *AuxiliaryConstructs::Templates* is the UML metaclass that can represent classes with template parameters.

AuxiliaryConstructs::Templates defines *Classifier* to apply the template associations that includes the metamodel to metaclass *Clases::Kernel ::Classifier* and its subclasses (such as Class, Collaboration, Component, Datatype, Interface, Signal, and Use Cases). In this profile, the metaclass of model element that has associated the parameters must be *Class*.

`<<QoSCharacteristic>>` includes the attribute `isInvariant`. `<<QoSDimension>>` includes the attributes `unit`, `direction`, and `statisticalQualifier`.

The base class of stereotype `<<QoSCategory>>` is *Package*. The package is a grouping of *QoS Characteristics* that provides support for the management of *QoS Characteristics*. *QoS Categories* are modeled in UML 2.0 with packages, and the packages include *QoS Characteristics* and other *QoS Categories*.

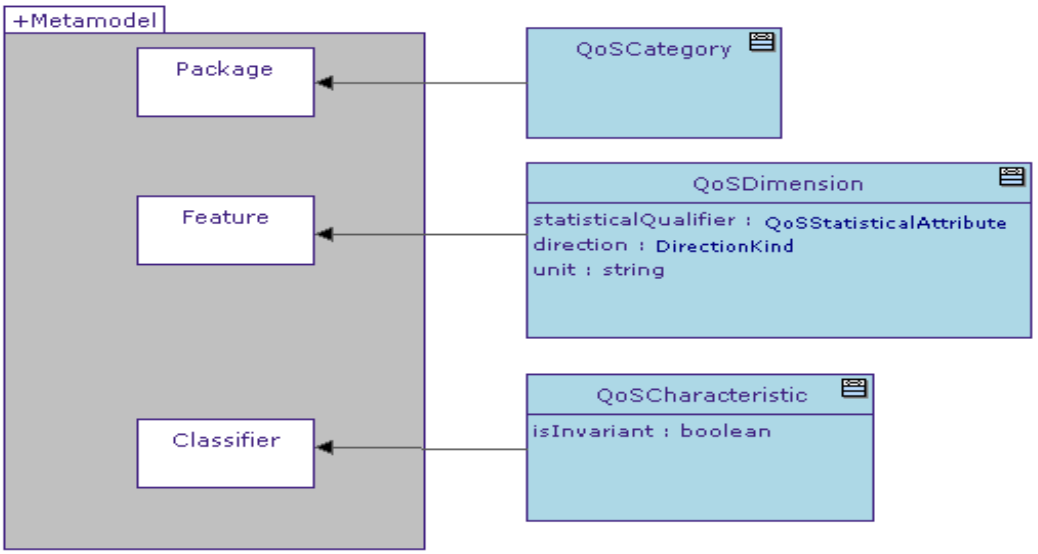

Figure 9.2 - QoS Characteristics Stereotypes

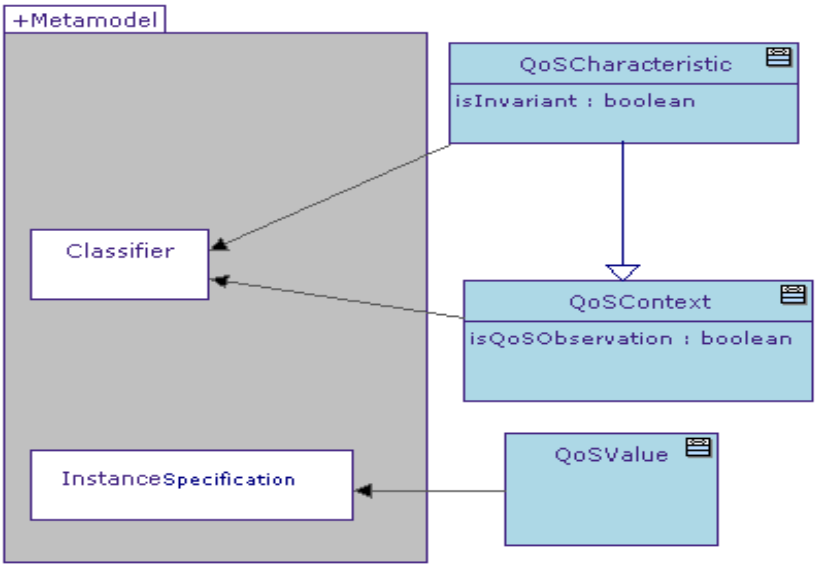

Figure 9.3 - QoS Values Stereotypes

The base class of stereotype <<QoSValue>> is *InstanceSpecification*. The model element annotated is an instance of a class annotated <<QoSCharacteristic>>. The *QoS Value* is implicitly a constraint whose slots fix the values of *QoS Dimensions*.

9.1.1 Well-formedness and Semantics

When a property is annotated with the stereotype <<QoSDimension>>, the class that includes the property must be annotated with the stereotype <<QoSCharacteristic>>.

```
context QoSDimension inv:
     self.base.class.
         stereotypes->includes(QoSCharacteristic)
```

The type of a <<QoSDimension>> represented with a property can be a <<QoSCharacteristic>>, a primitive type, or an enumerated type.

```
context QoSDimension inv:
   self.base.classifier.isTypeOf(Property) implies
           (self.base.classifier.isTypeOf(Enumeration) or
           self.base.classifier.isKindOf(PrimitiveType) or
           (self.base.classifier.isTypeOf(Class) and
            self.base.classifier.
              stereotypes->includes(QoSCharacteristic)))
```

The classes annotated with <<QoSCharacteristic>> and class parameters cannot have *QoS Value* instances.

```
context QoSValue inv:
   self.base.classifier->forAll(self.isKindOf(
            UML::AuxiliarConstructs::Templates::Classifier) implies
                  self.ownedSignature.inheritedParameter.size() = 0)
```

isQoSObservation identifies some kind of QoS monitoring. The *QoS Context* defines the quality values that are collected. The collected information includes all quality characteristics and context navigable from the original context excluded other *QoS Context* that are QoS Observations. When we can navigate from one context QoS Observation to another, we maintain references that allow to express end-to-end quality expressions and expressions that make reference to qualities of multiple elements. If a *QoS Context* with *isQoSObservation* true were used in expressions of multiple elements, we could not identify which is the element that we reference. Because of this, when *isQoSObservation* is true, *QoS Context* can only annotate one model element.

```
context QoSContext inv:
 let ns : Namespace = self in
 self.isQoSObservation implies
   Constraint.allInstances->select(
     self.stereotypes->isKindOf(QoSConstraint) and
     context = ns)->collect(element : Set(Element) |
       element.union(self.constrainedElement))-> size() = 1
```

Three types of modeling elements can represent quality dimension. The specific semantic depends on the specific modeling element that annotate <<QoSDimension>> stereotype. We identify three different modeling elements:

1. *Attributes*: Properties owned by the class that represent primitive values for the dimension.
2. *Navigation Properties*. Properties owned by association that allows the definition of dimensions based on other *QoS Charateristics*.
3. *Behavior Features*. This type of *QoSDimension* supports the description of dimensions with quality expressions (expressions that represent a quality dimension based on other dimensions).

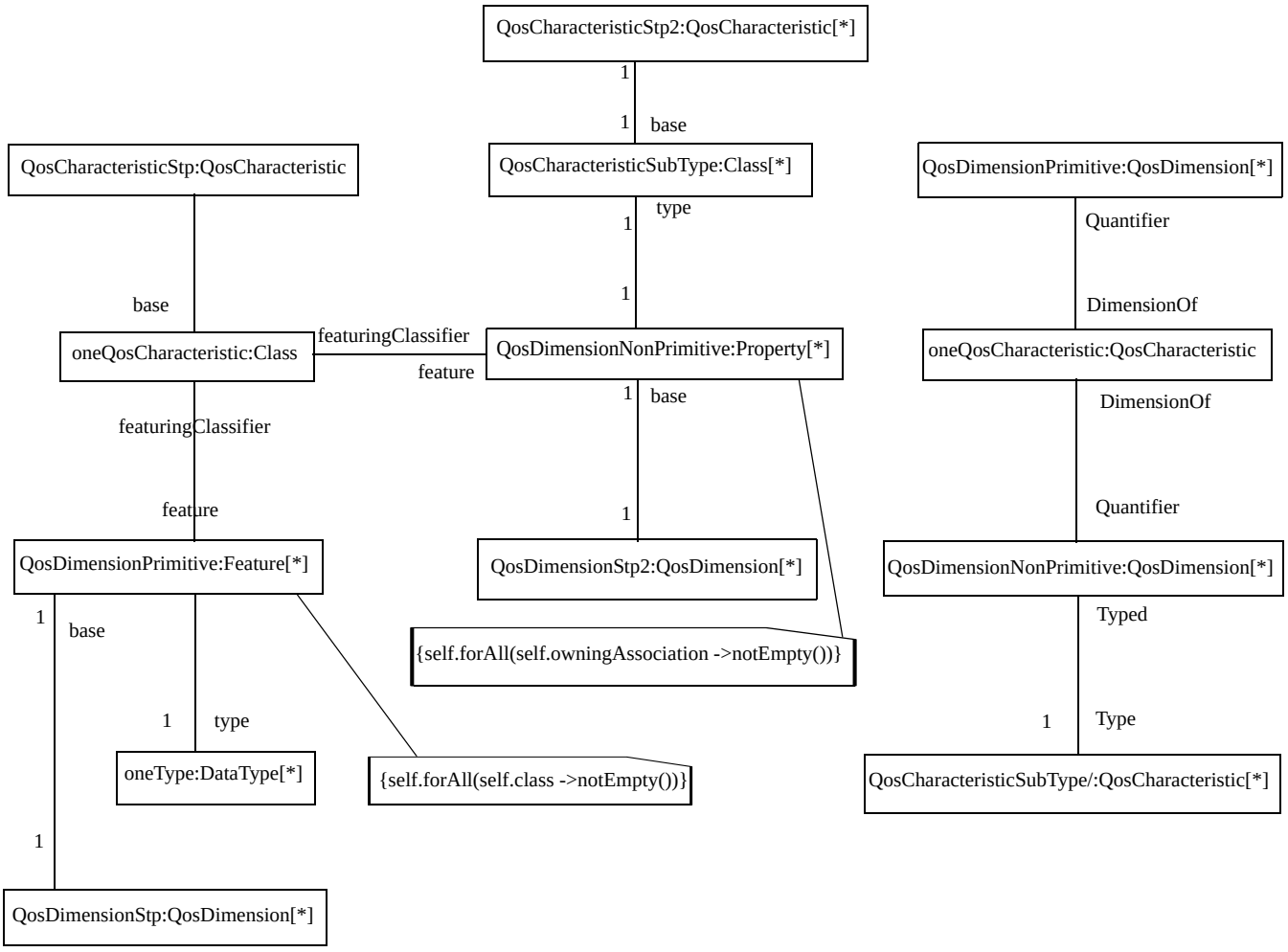

Figure 9.4 - (a) QoS Characteristics and Dimension in UML 2.0 (b) QoS Characteristics and Dimension in QoS metamodel

Figure 9.4 (a) includes a collaboration based on a set of roles that represent the definition of a *QoS Characteristic* and its dimensions. The diagram includes two types of dimensions (primitive and navigable). Both types of roles are roles with multiple instances. The attributes are owned by the class, and the navigation properties the feature is owned by the association. In the connections of this and the rest of composite structure diagrams we do not use the name of associations as classifier of the connector, because UML 2.0 does not include names in the associations. We use the property names, which include the associations of UML 2.0, in the connector end. Figure 9.4 (b) represents the equivalent elements in the QoS language that defines the QoS metamodel. The roles represent the QoS model elements that should be created to represent the equivalent in UML, and the reference roles represent referenced elements in the QoS model that are not created (e.g., the role `subCharacteristic` makes reference to some *QoSCharacteristic* that exists in this QoS model).

The UML 2.0 metaclass *Generalization* and the metaclasses of *Template* metamodel provide support to represent the association *Parent-Sub*, and metaclass *QoSParameter* of QoS metamodel. We do not propose explicit stereotypes to represent these concepts. We propose an implicit approach that reduces the model annotations. The generalization associations of two classes stereotyped with `<<QoSCharacteristic>>` represents the *Parent-Sub* association of QoS

metamodel. Figure 9.5 represents the correspondence between the *Parent-Sub* association expressed in UML 2.0 viewpoint and *QoS Characteristic* metamodel. Figure 9.6 describes the correspondence between the description of template in UML 2.0 and the parameters of *QoS Characteristics*.

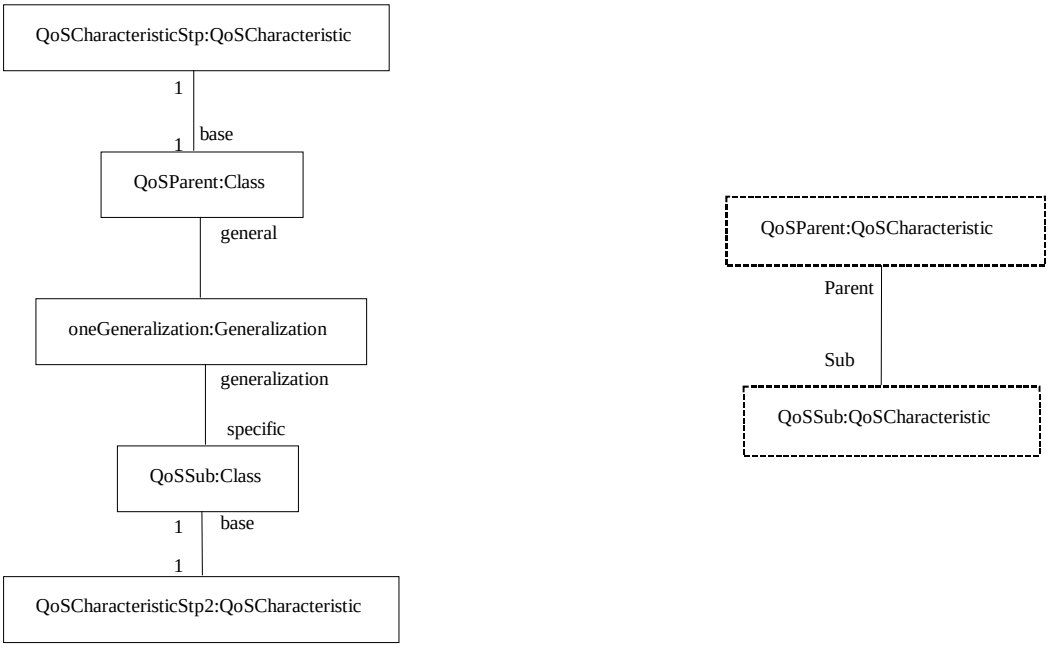

Figure 9.5 - (a) QoS Paren-Sub in UML 2.0 (b) QoS Parent-Sub in QoS metamodel

Figure 9.6 describes the correspondence between the description of template in UML 2.0 and the parameters of *QoS Characteristics*. Figure 9.6 (a) includes a template that represent the parameters description for a QoSCharacteristic and one binding from another characteristic. Figure (b) represents the characteristics equivalent in QoS metamodel that creates the binding references between characteristic and associate the parameters.

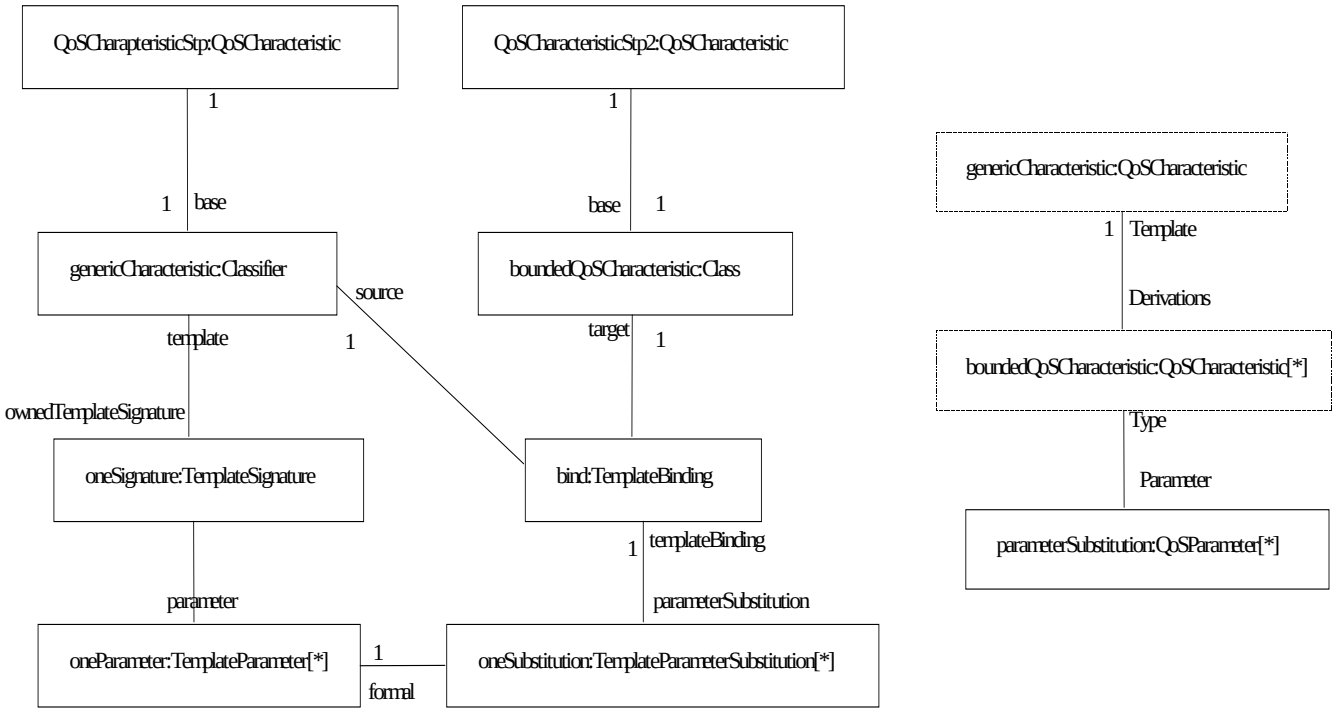

Figure 9.6 - (a) QoS Templates UML 2.0 (b) QoS Parameter in QoS metamodel

Figure 9.7 (a) includes the representation of a *QoS Value* in UML 2.0 and its relations with *QoS Characteristics* and slots. Figure (a) includes dimensions primitives and dimensions that reference other *QoS Characteristics*. The UML model element that represents the *QoS Value* is `InstanceSpecification`. UML 2.0 uses the metaclasses `Slot` and `ValueSpecification` for the description of attributes and their values. Figure (b) includes the representation of same concepts in QoS metamodel. Two different types of `QoSDimensionSlot` represent dimensions primitive or dimensions that make reference to other *QoS Characteristics*.

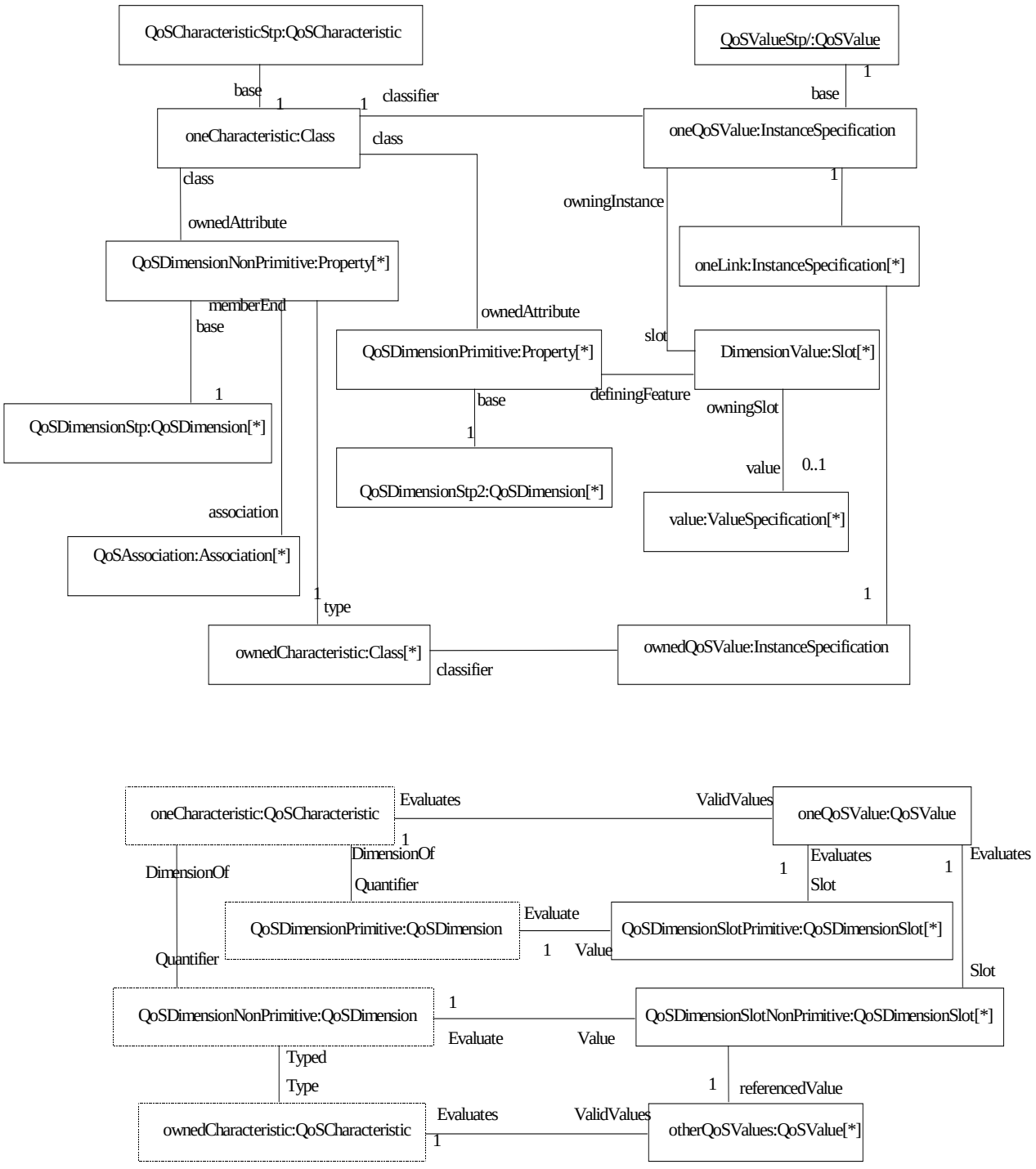

Figure 9.7 - (a) QoS Value in UML 2.0 (b) QoS Value in QoS metamodel

The base class of stereotype `<<QoSCapability>>` is *Package*. The package is a grouping of *QoS Characteristics* that provides support for their management. *QoS Categories* are modeled in UML 2.0 with packages, and the packages include *QoS Characteristics* and other *QoS Categories*. Figure 9.8 (a) describes the modeling of *QoS Categories* in UML 2.0 and Figure (b) the equivalent in QoS metamodel.

The base class of stereotype `QoSContext` is *Classifier*. A UML *Classifier* with `<<QoSCharacteristic>>` stereotype defines a *QoS Context* of QoS metamodel. The `<<QoSContext>>` stereotype is not required to define the context for this simple characteristic. *QoS Contexts* can reference other *QoS Contexts and QoS Characteristics*. Figure 9.9 describes the representation of a *QoS Context* with reference to other context and characteristics in UML 2.0 and the equivalent in QoS metamodel.

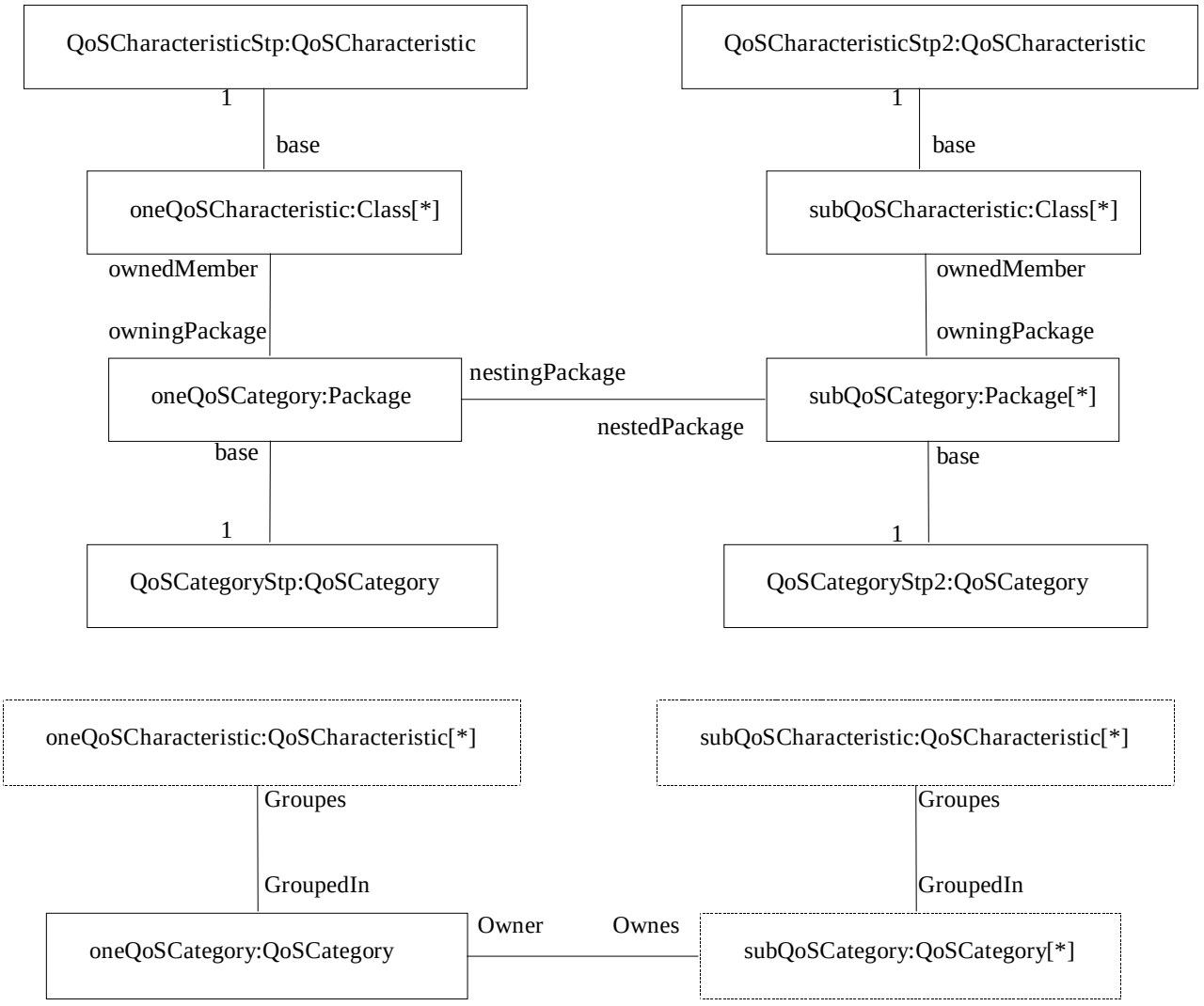

Figure 9.8 - (a) QoS Category in UML 2.0 (b) QoS Category in QoS metamodel

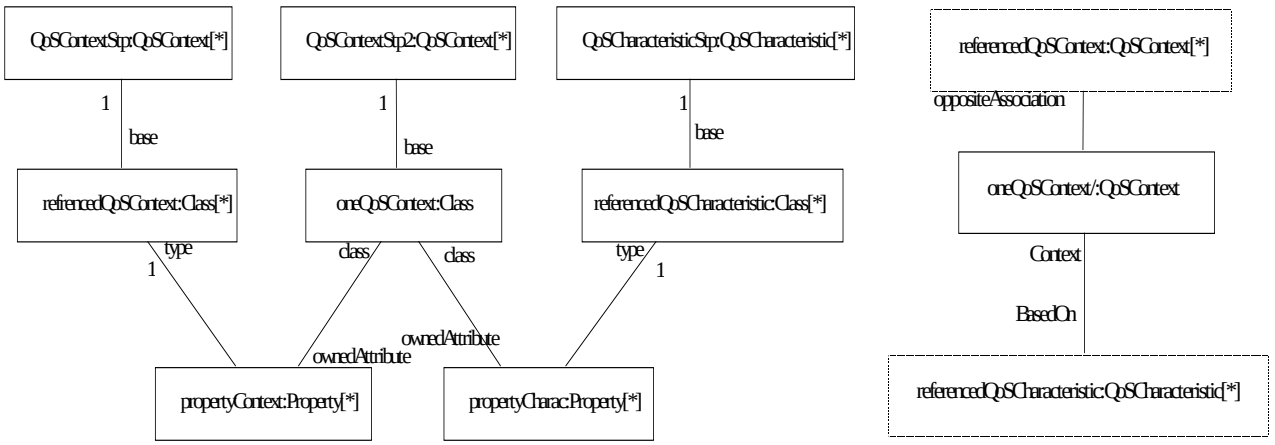

Figure 9.9 - (a) QoS Context in UML 2.0 (b) QoS Context in QoS metamodel

9.2 QoS Constraints Subprofile

QoS Constraint is an *abstract* stereotype (there is not instances of this stereotype); we use it for structural purposes in the profile. The QoS constraint stereotypes are: `<<QoSRequired>>`, `<<QoSOffered>>`, and `<<QoSContract>>`. The attribute *Qualification* specifies the strictness of the constraint. The values are: *Guarantee*, *Best-Effort*, *Threshold-Best-Effort*, *Compulsory-Best-Effort*, and *none*.

There are two methods for the specification of *QoS Constraints* in UML:

1. UML constraint with stereotype `<<QoSRequired>>`, `<<QoSOffered>>`, or `<<QoSContract>>`. These constraints include OCL expression whose context is a *QoS Context* classifier. The expression in the OCL expression limits the allowed values of *QoS Characteristics* associated to the *QoS Context*. The properties a*llowedValue* and *logicalOperator* are not used.

2. UML *Dependency* relationship with stereotype `<<QoSRequired>>`, `<<QoSOffered>>`, or `<<QoSContract>>`. A model element (client) depends on a *QoS Value* (supplier). The attribute *logicalOperator* specifies the relationship between values when there are multiple values (because the dependency has multiple suppliers, or because the same client has multiple dependencies to different *QoS Values*). The values for *logicalOperator* are: *and*, *or*, and *none*. The attribute a*llowedValue* is not used. *logicalOperator* represents the logical relation when there are multiple *QoS Values* in the dependency. When the *logicalOperator* is *and*, the quality required or provided must achieve all *QoS Values*. When the operator is *or*, the quality provided or required must have at least the value of one *QoS Value*.

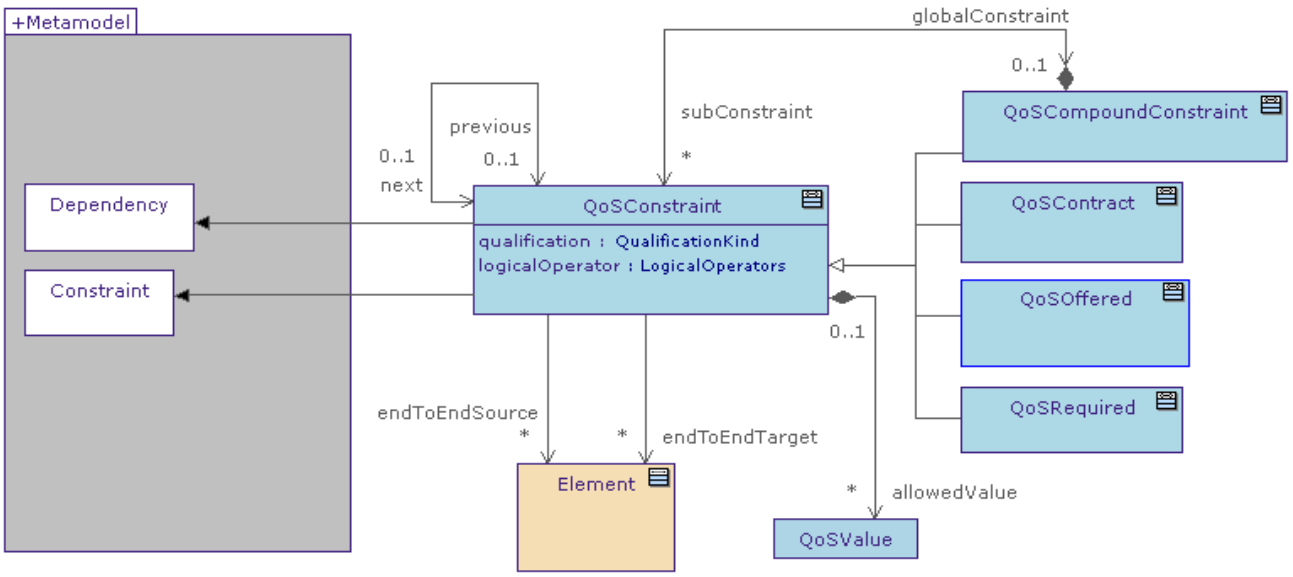

Figure 9.10 - QoS Constraints Stereotypes

9.2.1 Well-formedness and Semantics

The supplier of a UML dependency annotated with stereotype `<<QoSRequired>>`, `<<QoSOffered>>`, or `<<QoSContract>>` must be an *InstanceSpecification* annotated with stereotype `<<QoSValue>>`.

```
context QoSConstraint inv:
          self.base.isKindOf(Dependency) implies
          self.base.supplier->forAll(
            self.isKindOf(InstanceSpecification) and
            self.stereotypes->includes(QoSValue)
          )
```

The context of constraints annotated with stereotype `<<QoSRequired>>`, `<<QoSOffered>>`, or `<<QoSContract>>` must be a *Classifier* with stereotypes `<<QoSContext>>` or `<<QoSCharacteristic>>`.

```
context QoSConstraint inv:
         self.base.isKindOf(Constraint) implies
          self.context->forAll(self.stereotypes->
                  forAll(self.isKindOf(QoSContext))
          )
```

In Figure 9.10 *QoSConstraint* extends metaclasses *`Element`*, *`Constraint`*, and `Dependency`. `Element` is superclass of *`Constraint`* and `Dependency`. But these extensions represent the three methods for the specification of *QoS Constraints*. The semantic of the constraints depends on the model element that annotate the UML constraint (method 1), or the client of the dependency (method 2). Examples of modeling elements annotated with *QoS Constraints* are actor, class, component, node, object, subsystem, use case, and instance. The constraints limit the quality of services that provide the modeling element. The annotation of interface elements limits the quality of its implementators, or

models the quality that requires its users. The annotation of modeling elements such as associations, connectors, and ports represent the quality that requires the clients that use the specific services provided in these relationships, and the quality provided.

Figure 9.11 (a) represents a *QoS Constraint* in UML with a UML constraint. The constraint is annotated with a *QoS Constraint* stereotype. The context of the constraint is a *QoS Context* or a *QoS Characteristic*. Figure 9.11 (b) represents the equivalent concepts in QoS metamodel.

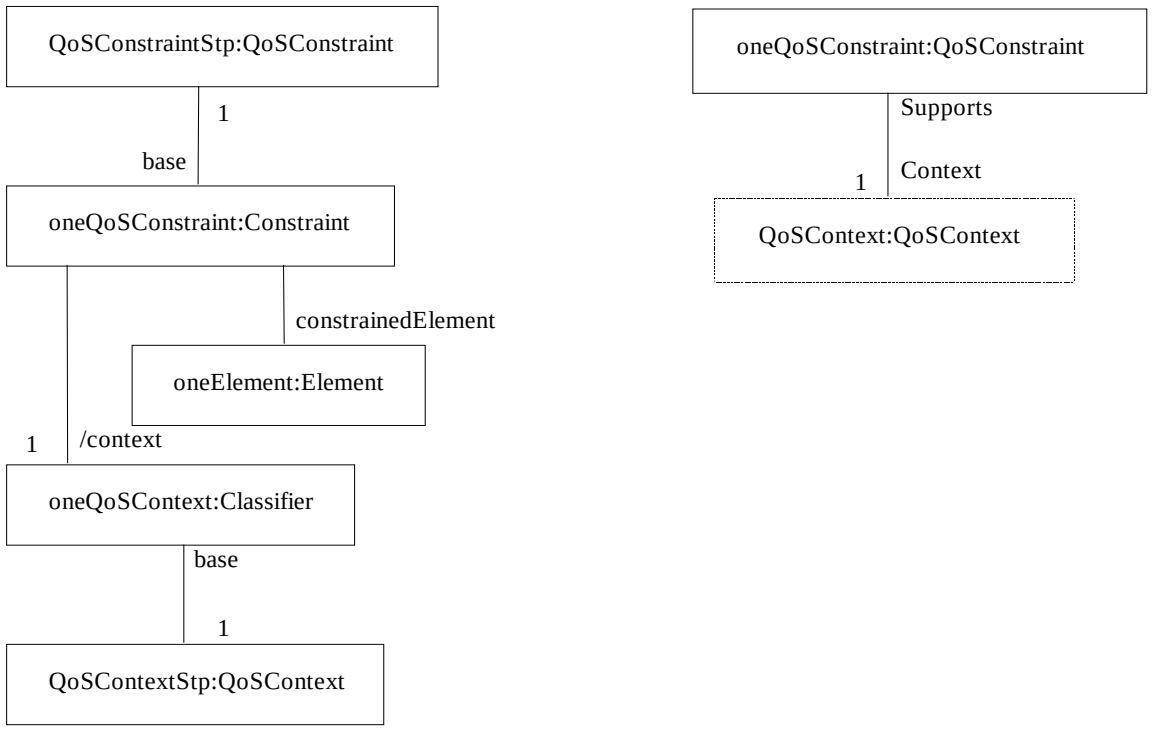

Figure 9.11 - (a) QoS Constraints in UML 2.0 (b) QoS Constraint in QoS metamodel

9.3 QoS Behavior Subprofile

The base class of stereotype <<QoSLevel>> is *State*. States with stereotype `<<QoSLevel>>` model the quality state of model elements like classes or components. The *QoS Level* states can have associated *QoS Constraints*.

The base class of stereotype `<<QoSTransition>>` is *Transition*. It models the quality transition of software elements associated to the state machine where it is included. These transitions connect *QoS Level* states (states with stereotype `<<QoSLevel>>`), and can have associated a *QoS Adaptation Process*. These transitions have associated the implicit event that occurs when the *QoS Allowed Spaces* of source state becomes false, and the *QoS Allowed States* become true for the target state. This event represents the *QoS Level Change* metaclass.

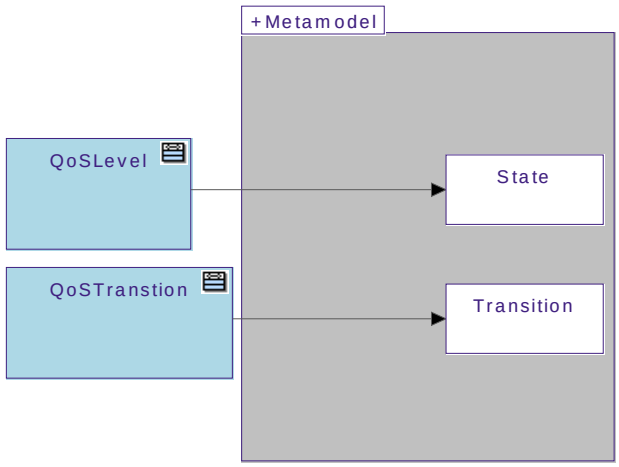

Figure 9.12 - QoS Behavior Stereotypes

9.3.1 Well-formedness and Semantics

The transitions that connect two *QoS Level* states are *QoS Transitions*:

```
context QoSLevel inv:
 self.base.outgoing->union(self.base.incoming)->
                    forAll(self.stereotypes.includes(
                                              QoSTransition))

context QoSTransition inv:
 self.base.source->union(self.base.target)->
         forAll(self.stereotypes.includes(QoSLevel))
```

9.4 Integration of General Resource SubProfile

Figure 9.13 includes the stereotypes in GRM subprofile of SPT that provide support for the identification of dependencies of resources. These dependencies can be annotated with quality constraints and characteristics to constraint the quality of services that provide the resource. These stereotypes are specified in Chapter 4 in the standard [20].

GRM subprofile does not include directly any stereotype for the identification of resource. The concept is described at conceptual level, but it does not include an UML extension. The subprofile that uses this subprofile (*Scheduling Analysis* and *Performance Modeling*) introduces specific resource stereotypes, but for analysis purposes (they include attributes that represent analysis results and specific values). To represent a general resource we introduce a new stereotype that identifies the resources. This stereotype provides support for the description of QoS provided or required from resources. Figure 9.14 includes this new stereotype.

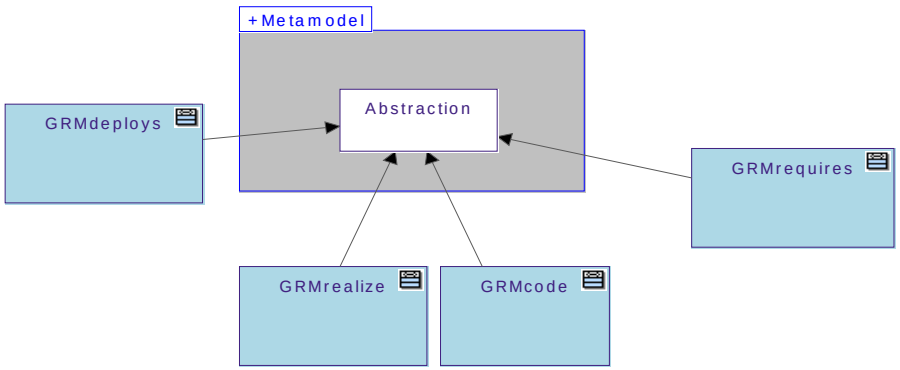

Figure 9.13 - GRM Subprofile Stereotypes

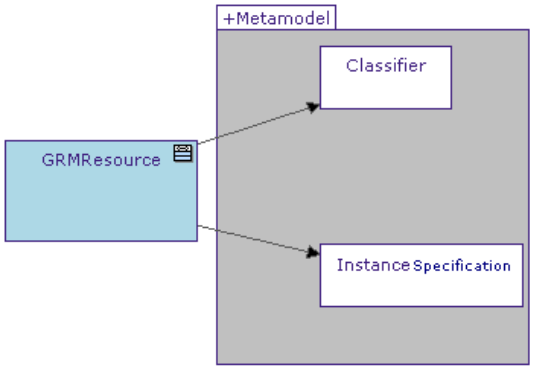

Figure 9.14 - Resource Stereotype

10 QoS Catalog

General QoS *Characteristics* and *Categories* can be reused in different projects and domains. They include characteristics whose quantification dimensions are not problem specific. A general QoS catalog includes a set of general characteristics and categories that are not specific of projects or domains.

The analysis for identification of characteristics required in a project is a complex process that, as other analysis process, requires some experience and a good knowledge of types of non-functional requirements of the domain. But the quality models for specific domains are easy to reuse, because these models have a very few changes between projects, the quality quantification is practically the same in most of the projects, and some of the characteristics (e.g., the characteristics for the quantification of qualities of resources and infrastructures) are associated to types of resources and platforms common on several projects of the same domain.

A quality model is easy to reuse in the specification of non-functional properties of different projects. A common quality model is a good candidate to reduce cost of specification of non-functional properties by reusing the quality models.

This section introduces a general quality model independent of domains and problems. This general catalog can be extended with domain specific characteristics to define a domain QoS catalog, and each domain QoS catalog would be specialized for specific projects. The objective of this section is not creating a standard of reference model for general QoS characteristics. Several standards have been adopted for these purposes. These standards have specific purposes, and in some cases they address the problem from the software metric perspective (quality attributes for the evaluation of software, not for the qualification of services), and in all cases there is not modeling language used for the representation of characteristics. The objective of this section is to define QoS characteristics, particularly those characteristics important to real-time and high confidence systems, which describe the fundamental aspects of the various specific kinds of QoS and create a common framework that relates all of them. These characteristics will be extended or specialized for specific domain and projects.

We have used different references to identify the *QoS Characteristics* and *Categories* that we are going to introduce. None of them try to use UML as specification language, or try to integrate the characteristics with UML models. The main references are [14][4][13].

10.1 General QoS Categories

Figure 10.1 includes a set of general QoS categories:

- *Performance*: Performance makes reference to the timeliness aspects of how software systems behave, and this includes different types of *QoS Characteristics*: latency and throughput. Sometimes it refers to the relationship between the services provided and the utilization of resources: memory and CPU consumptions.
- *Dependability* [16]: Dependability is the property of computer systems such that reliance can justifiably be placed on the service it delivers. It includes QoS Characteristics such as: availability, reliability, safety, and integrity.
- *Security.* This capability covers different subjects such as the protection of entities, and access to resources. *QoS Characteristics* included in this capability are access control and confidentiality.
- *Integrity:* Sometimes the service provided is not the service expected, but it is functionally correct and does not directly produces faults itself. A specific case are the levels of error or accuracy that are different in the service provided and the service expected.
- *Coherence:* Coherence includes characteristics about concurrent and temporal consistency of data and software elements.

- *Throughput*: Throughput refers to the number of event responses handled during an observation interval. These values determine a processing rate.
- *Latency*: Latency refers to a time interval during which a response to an event must arrive.
- *Efficiency*: The capability of the software to produce their results with the minimum resource consumption.
- *Demand:* Demand is the characterization of how much of a resource or a service is needed.
- *Reliability*: The capability of the software product to maintain a specified level of performance when used under specified conditions.
- *Availability*: Availability is the capability of the software product to be in a state to perform a required function at a given point in time, under stated conditions of use. Externally, availability can be quantified by the proportion of total time during which the software product is in an up state.

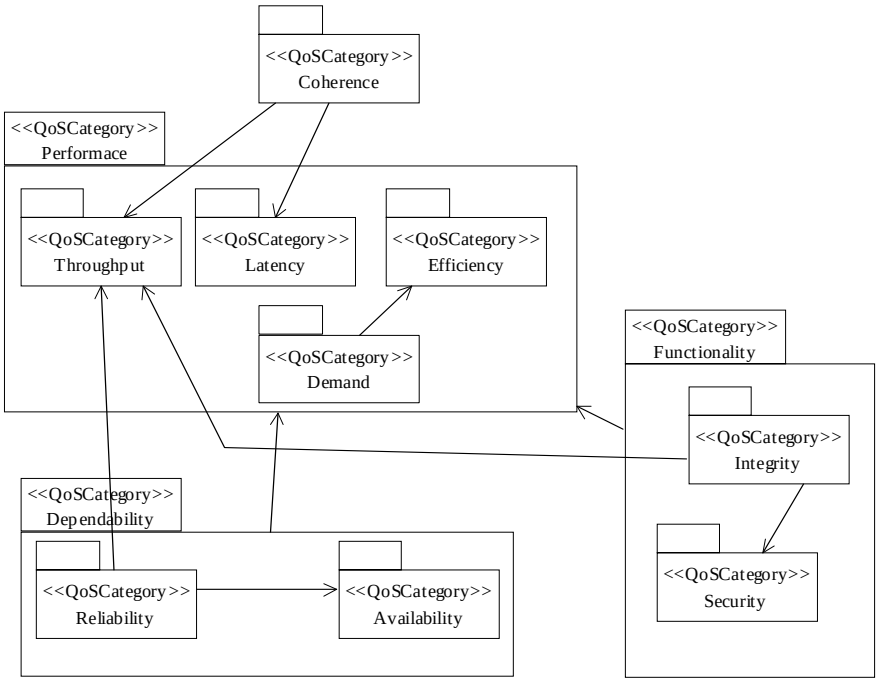

Figure 10.1 - Quality Categories

10.2 Throughput Characteristics

Figure 10.2 includes a model for the description of different types of application of throughput concept. An abstract *QoS Characteristics* (*throughput*) represents the throughput in general, during an interval of time and a rate, whose units or direction are not defined (because it is abstract). This diagram considers three types of throughputs: *input-data-throughputs* represents the arrival rate of user data input channel, software or hardware, averaged over a time interval. The rate unit for this throughput is bit/sec, and the direction of this dimension is increasing. *communication-throughput*

represents the rate of user data output to a channel averaged over a time interval. The units and direction of rate are the same as *input-data-throughput*. *processing-throughput* represents the amount of processing able to be performed in a period of time. The unit of rate is instructions/sec and the direction increasing.

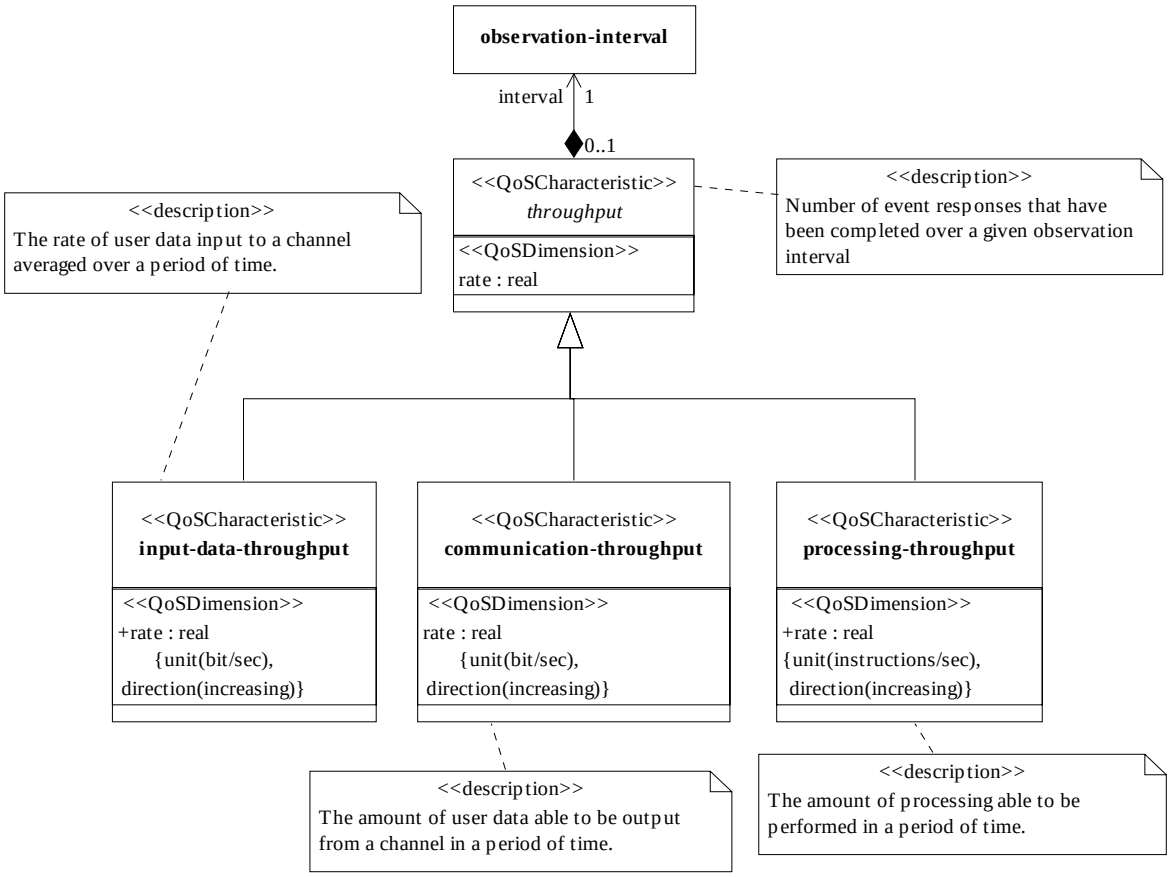

Figure 10.2 - Throughput Characteristics

10.3 Latency Characteristics

The package *Latency* includes two characteristics for the description of latencies. The characteristic *latency* is based on general dimension for the description of latencies for any kind of software elements. The characteristic *turn-around* is specific for the description of the absolute limit on time required in fulfilling a job task or service, or to represent the time required to perform a specific task, in the worst case. The turn-around is based on the description of the instant of request and the instant of result. The operation *turn-around-value* represents the difference and the direction is decreasing (the latency quality improves when the difference decreases). The other characteristic uses other types of dimensions, it refers to a time interval during which the response to an event must be executed. The response event may not be associated to the end of the task that starts the event. The time interval defines a response window with a minimum and maximum ending time. The jitter specifies the maximum variation in the time a computed result is output from cycle to cycle. Its worst value must be less than the window, in some cases the difference from cycle to cycle is known and less than the window. The criticality represents the importance of the event to the system.

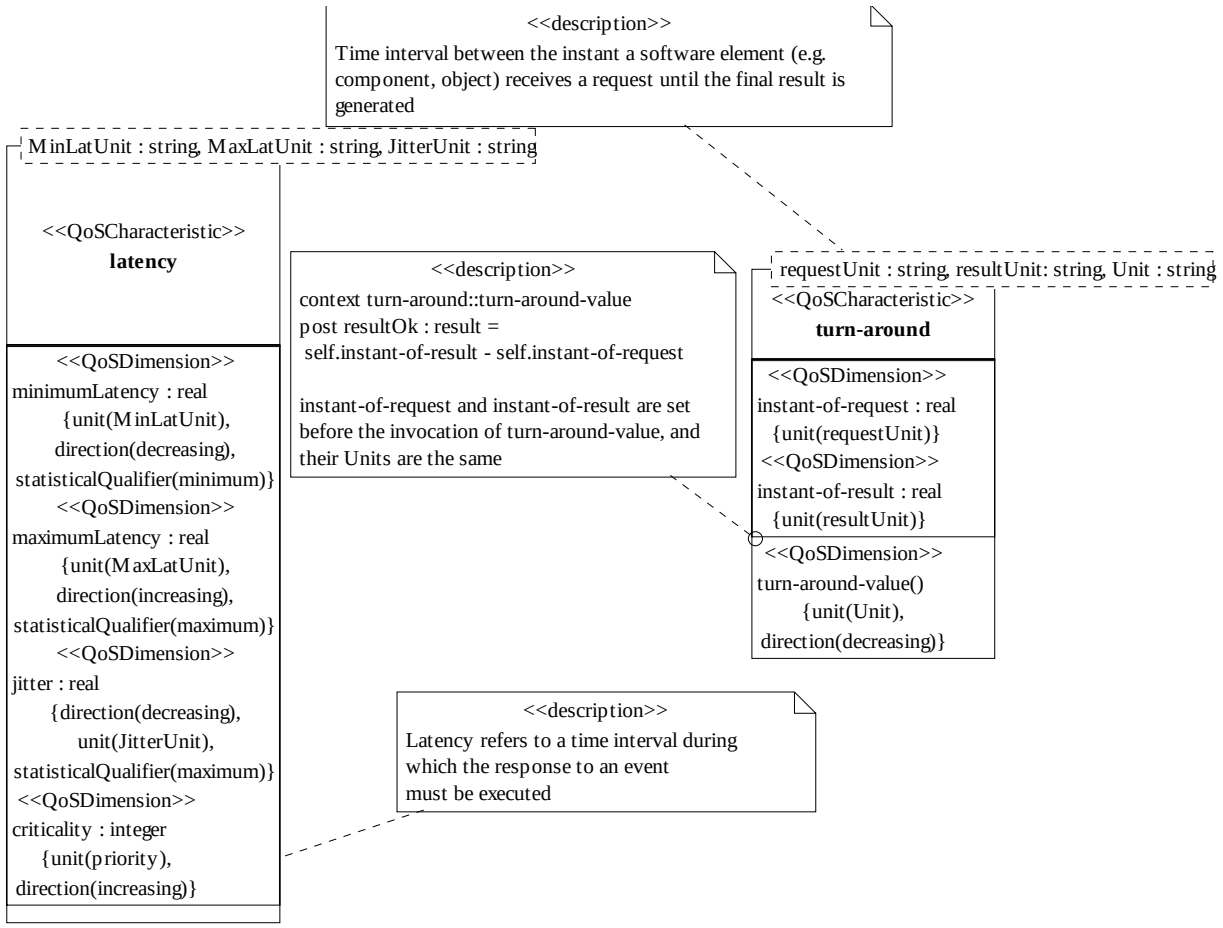

Figure 10.3 - Latency Characteristics

10.4 Efficiency Characteristics

The efficiency characteristics allow representing the execution time requirements for responding to each event.

Figure 10.4 includes a general characteristic for the representation of resource request, but we need to know the specific type of resource to concrete the units of resource requested. The *resource-utilization* characteristic only describes the utilization in a single action. The demand characteristics reuse this characteristic to describe general demands of resources.

Specializations of *resource-utilization* describe the utilization of computation, communication, and memory resources. They specialize the units of resource demand for the specific type of resource.

The package includes a general characteristic for the specification of QoS policies. This characteristic only includes some policies specific of real-time and QoS IP systems.

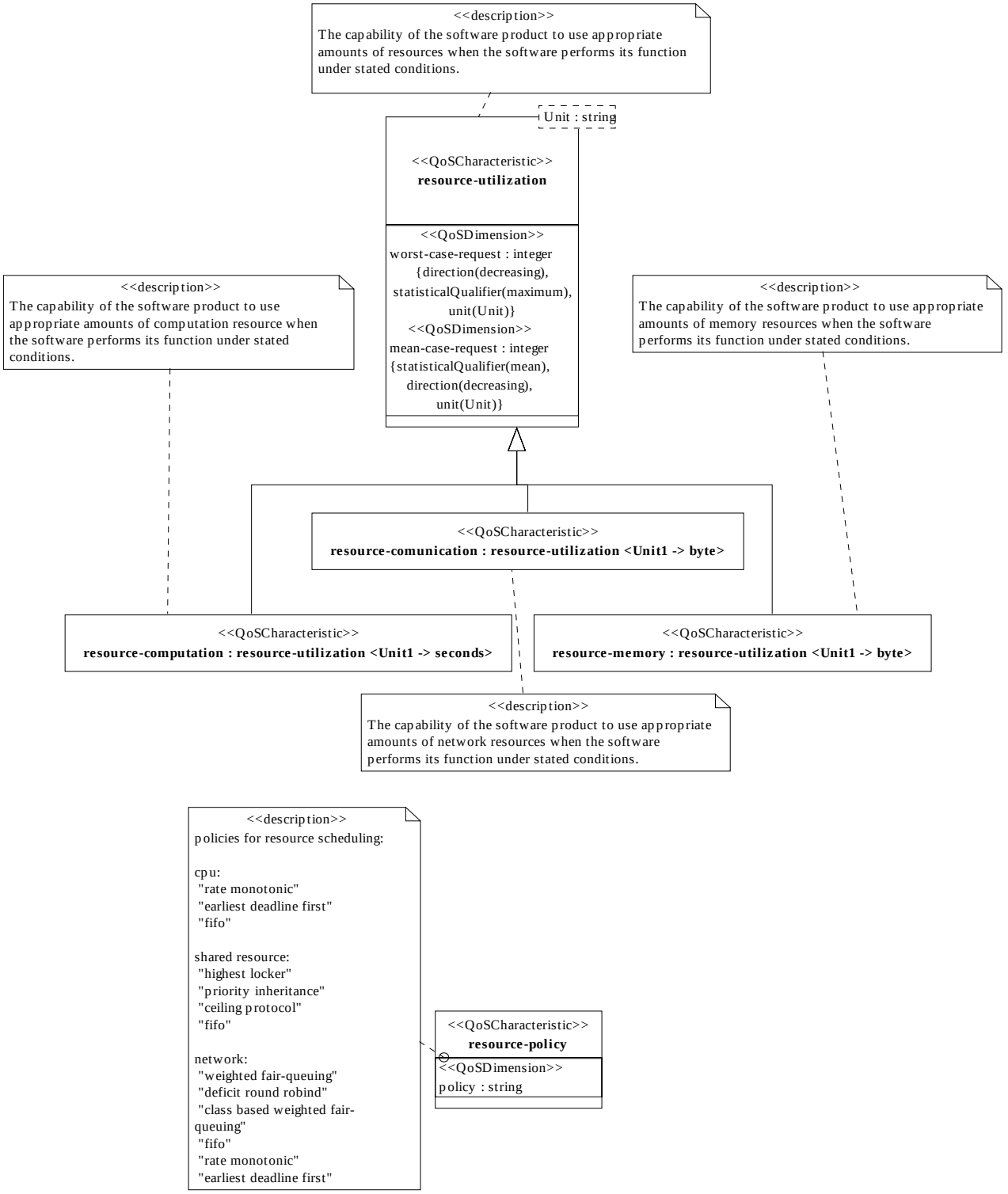

Figure 10.4 - Efficiency Characteristics

10.5 Demand Characteristics

The demand characteristics combine the *resource-utilization* characteristics with arrival patterns characteristics for the description of the amount of resources needed. The types of arrival pattern (periodic, irregular, bounded, bursty, and unbounded) and their dimensions define *arrival-pattern*. The dimensions are the interval (period of pattern arrival), jitter (the difference of pattern arrival from cycle to cycle), and burst size (the maximum number of occurrences in the time interval).

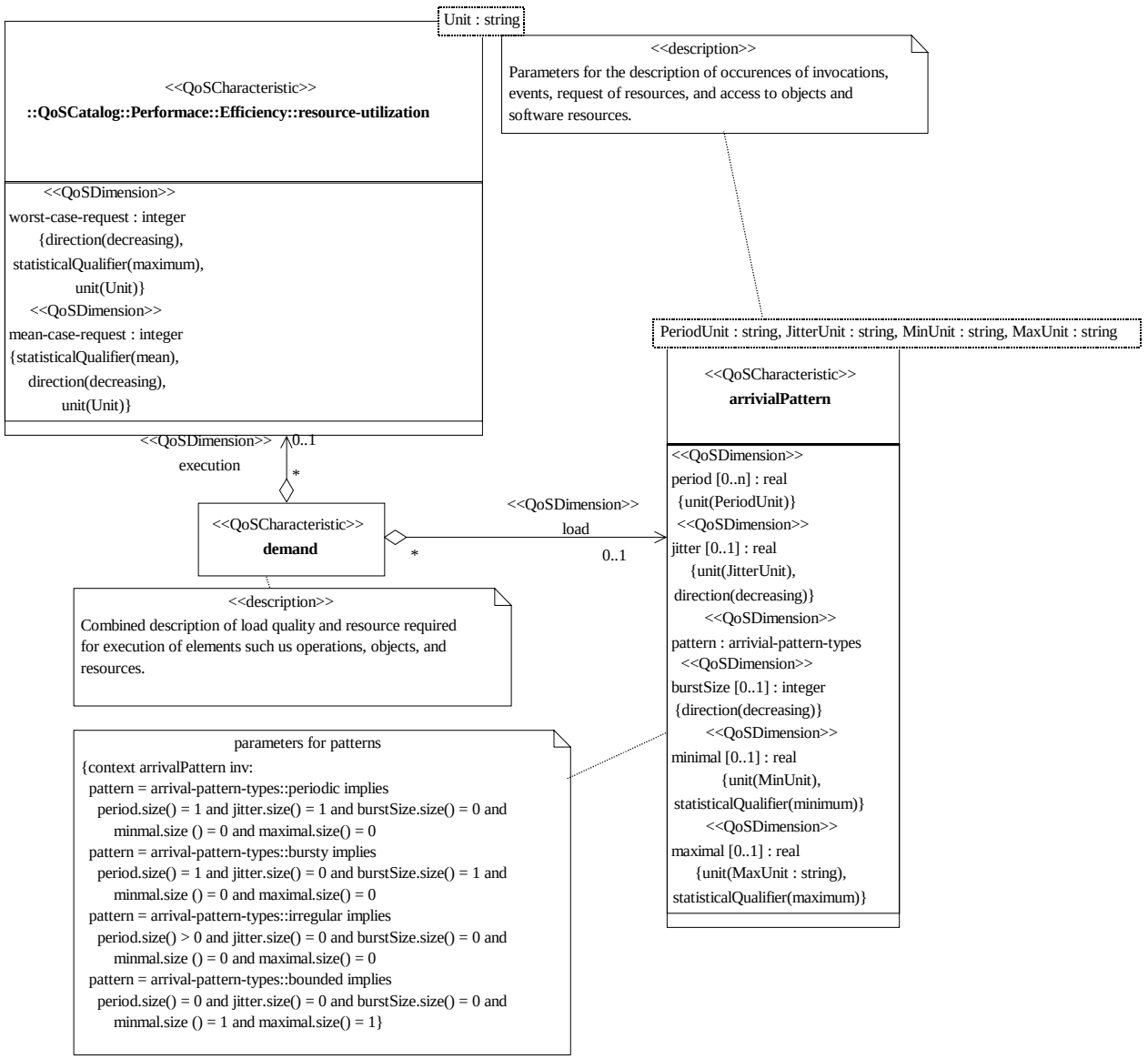

Figure 10.5 - Demand Characteristics

10.6 Integrity Characteristic

This *QoS Category* includes *QoS Characteristics* that describe allowed differences between the functional result expected and functional results provided. The main characteristic in this category is the *accuracy* that is another example of characteristic that can be dimensioned in different ways. Accuracy is a *QoS Characteristic* of concern to the user, for whom this characteristic refers to the integrity of the user information only. (The integrity of headers and similar protocol control information may be the subject of other characteristics). The accuracy characteristic is specialized in many ways, including addressing error, delivery error, residual error, etc. Figure 10.6 includes the first QoS characteristic for the description of the accuracy. It includes a dimension that expresses the maximum allowed difference between the expected result and the result provided. It is a parameterized characteristic, because the type of the dimension depends on the type of the results that we want to qualify. We must create an instance of this template with the specific type (e.g., real, integer, and application types), which will be used as characteristic.

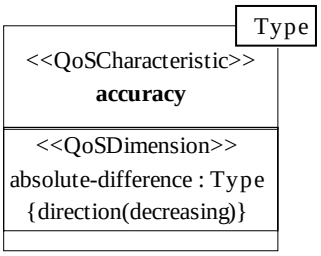

Figure 10.6 - Accuracy Error Characteristics

Figure 10.7 represents different types of accuracies, all of them in terms of accuracy errors. All of them include as dimension the probability of error occurrence. *transfer-integrity* and *establishment-error* include an additional dimension (*observation-interval*) that specifies the temporal interval of the error occurrence. The different types of errors are:

- addressing error: An incorrect choice of address(es) used for delivery of data.
- transfer error: The incorrect transmission of an amount of data.
- resilience: The ability to recover from errors.
- transfer integrity: The amount of data transferred in a time interval without error.
- establishment error: Inability to establish, within a specified time window, a connection or association that was requested.
- recovery error: Inability to recover from an error condition.

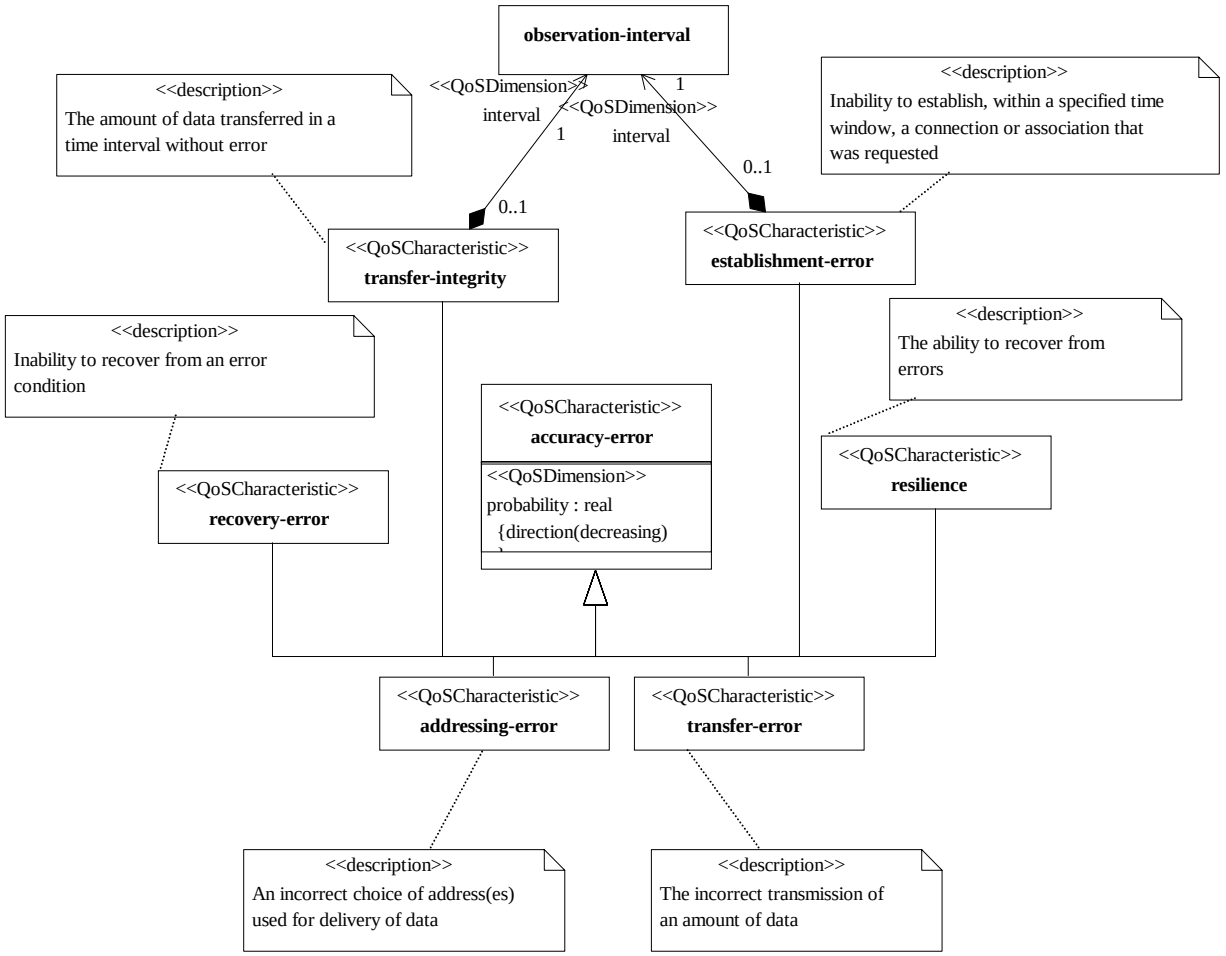

Figure 10.7 - Accuracy Error Characteristics

10.7 Security Characteristic

Security characteristics are another example of characteristics that can be dimensioned with different types of parameters (e.g., type of keys, types of encryption algorithms). We have used three general characteristics. The *access-control* specifies the control policy used in the access to the service. The policy has associated a security level. The *protection* is a more general characteristic for services that can have associated more than one access control method, and includes the probability of control break. Another characteristic (*safety*) is used to qualify the security level that an element provides. In this case we do not provide details about the methods of security used.

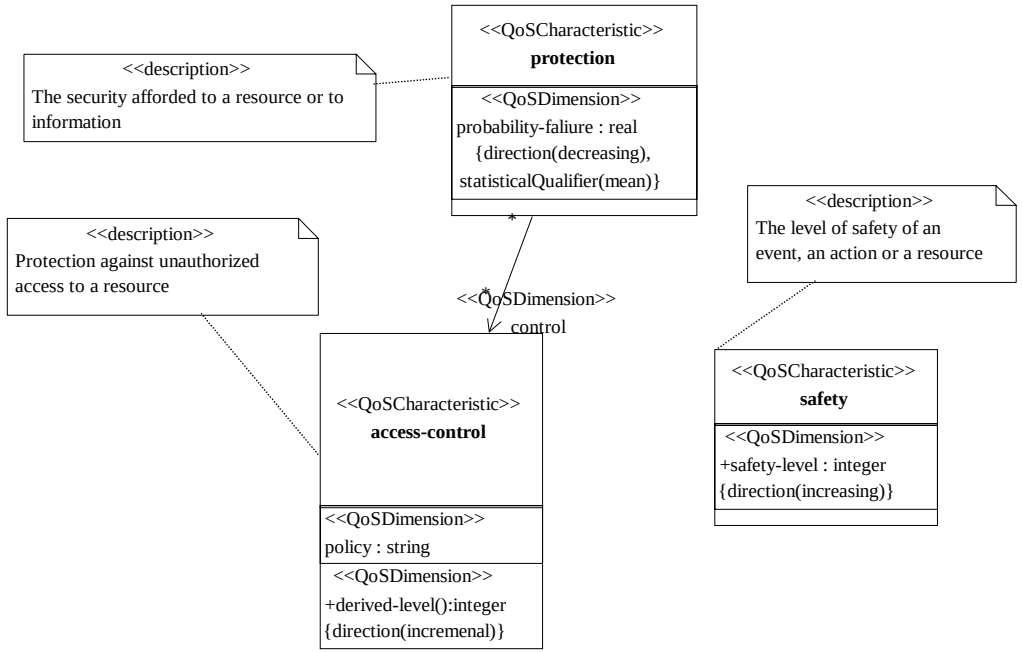

Figure 10.8 - Security Characteristics

10.8 Dependability Characteristics

The dependability includes several characteristics such as availability, reliability, safety (from a reliability perspective), confidentiality, and maintainability. We have paid special attention to the characteristics that qualify the services.

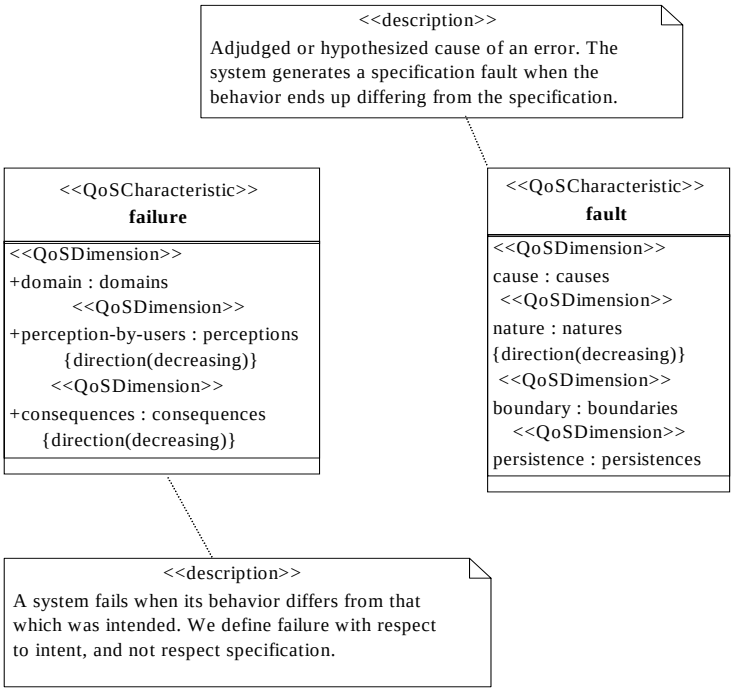

Figure 10.9 - Fault and Failure Characteristics

The impairments to dependability include the fault, error, and failure. The faults occur inside the system and the failures are observed in the environment [11]. The failures are evaluated with respect to intent, not the specification. The failures are quantified with three enumerated attributes: domain (value failures, timing failures), perception-by-users (consistent failures, inconsistent failures), consequences (benign failures, catastrophic failures). Figure 10.9 includes the characteristics.

Figure 10.10 includes characteristics for the description of *availability* and *reliability.* The availability is a measure of readiness for usage. There are different parameters for its evaluation; in this solution we use the *mean-time-between-failures* and the *mean-time-to-repair.* And the availability is calculated in the operation *availability-value*.

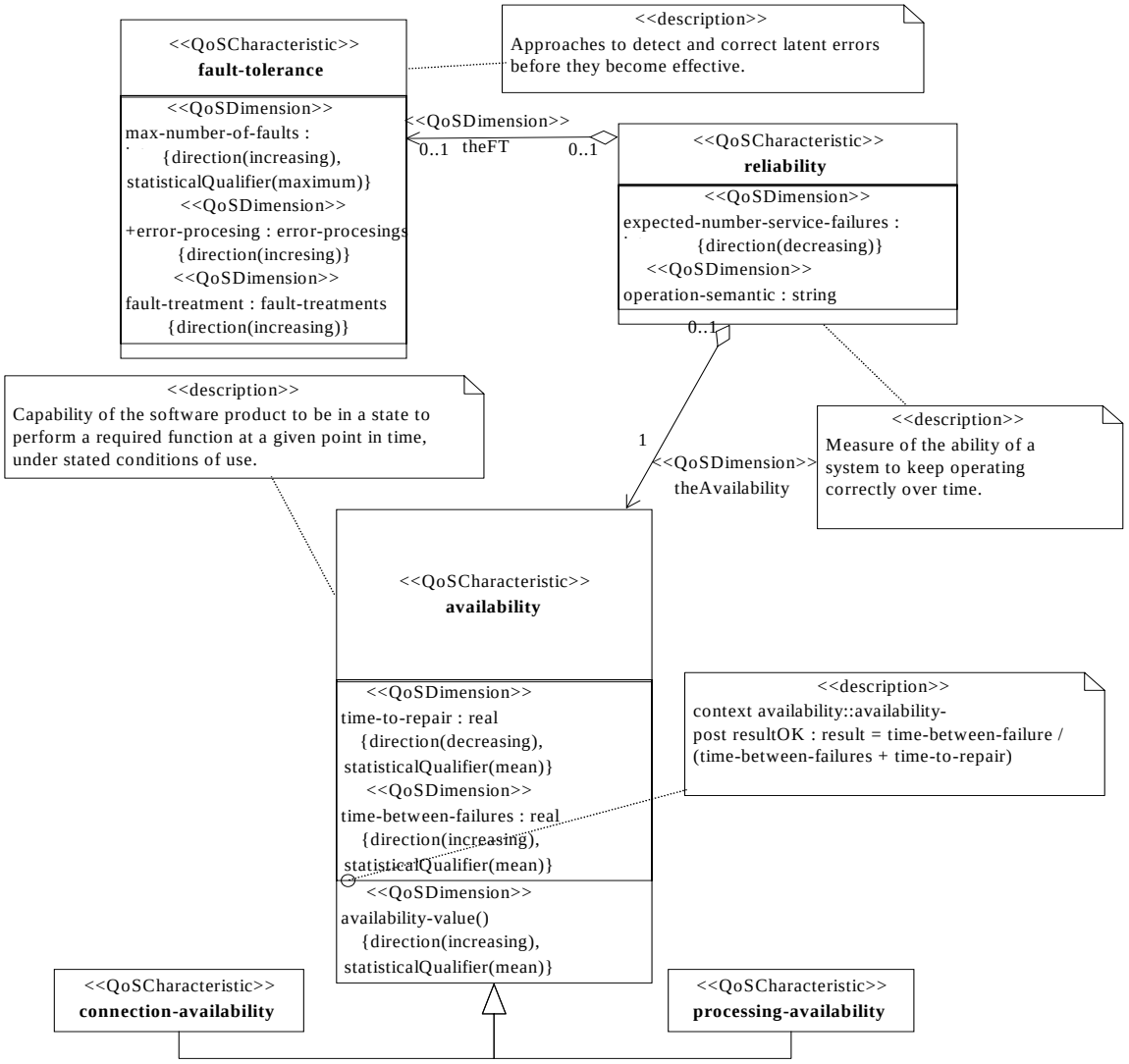

Figure 10.10 - Reliability and Availability Characteristics

Fault-tolerance are solutions to mitigate the reliability problems. In this approach we model the *fault-tolerance* characteristic with three dimensions: *maximum-number-of-faults* (the maximum number of faults supported), *error-processing* (what is done at removing errors: detection, diagnosis, backward recovery, forward recovery, and compensation), and *fault-treatment* (fault diagnosis and passivation: diagnosis, passivation, reconfiguration).

The *reliability* is the ability of a system to keep operating over time. The *availability* gives one dimension of reliability, and the *fault-tolerance* one possible solution to mitigate the reliability problems. The expected-number-of-failures is an estimation of the number of failures that the service generates, and the *operation-semantic* is the type of semantic that we can expect (only one time execution, at least one time execution, at most one time execution, none).

Maturity and *recoverability* are two specific types of reliability. The maturity size is the capacity of the software to avoid failures caused in the software itself. And the recoverability is the capacity to re-establish its adequate level of performance with minimum loss of data. The maximum number of errors in an interval evaluates the *maturity.* The time to recover and the time required to restart the system are the dimension of *recoverability.*

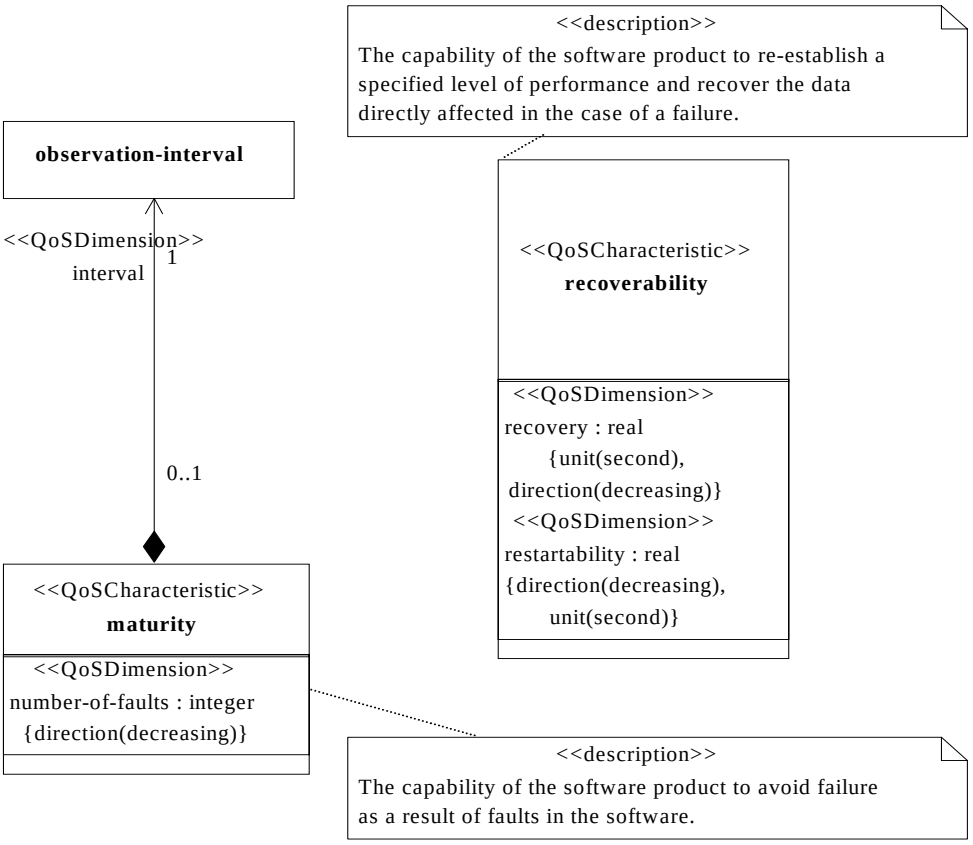

Figure 10.11 - Maturity and Recoverability Characteristics

10.9 Coherence Characteristic

The coherence includes characteristics for the evaluation of the concurrent and temporal consistency of data and functions.

The characteristic *coherence* in Figure 10.12 indicates whether an action has been performed on each entity (data item, value, etc.) in a list within a given time window. In this solution the characteristics represent the probability and the interval, the constraint attaches the characteristics to the set of elements.

10.10 Scalability Characteristic

Sometimes the same service is not produced with the same quality level when the number of software element increases. The capacity of software elements is limited to a minimum and maximum number of elements. For example, the minimum and maximum number of elements in a list is limited, and the quality of services over the list depends on the number of elements. Figure 10.13 includes a simple characteristic for the general description of scalability. The operation *cost-per-unit* includes a number as argument that represents the number of units, and returns a value that represents the cost for the application of services for this number of units.

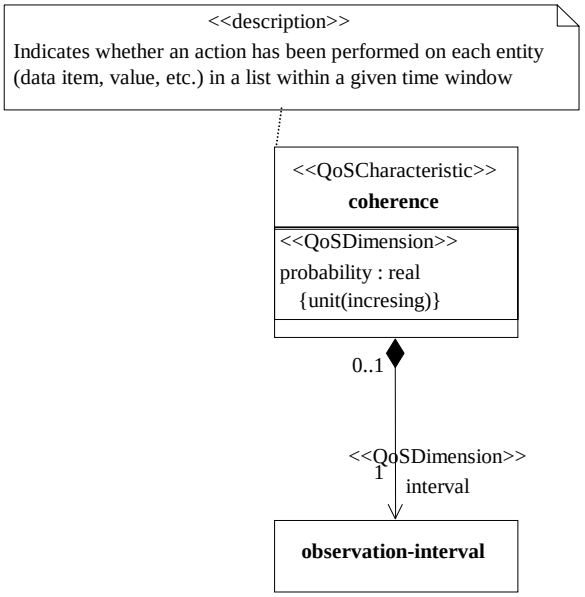

Figure 10.12 - Coherence Characteristics

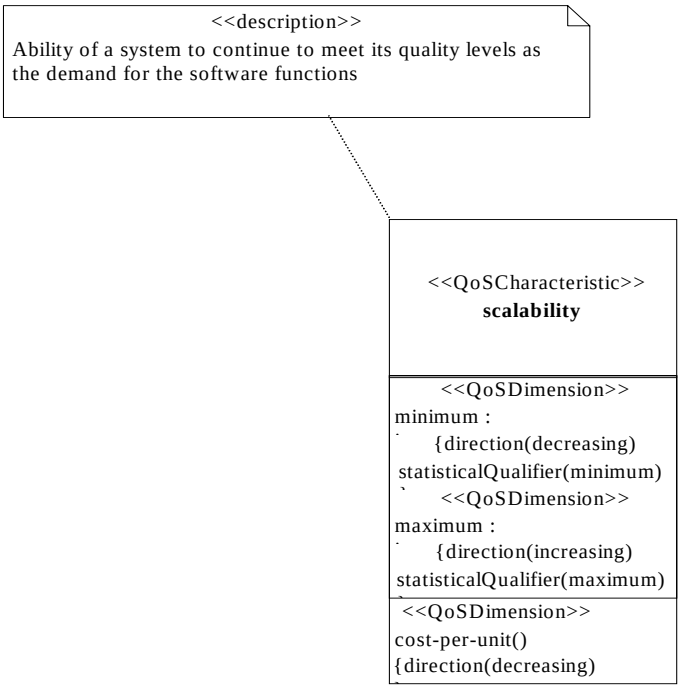

Figure 10.13 - Scalability Characteristics

11 Risk Assessment

This metamodel defines an abstract language for supporting model-based risk assessment. The aim of model-based risk assessment is to integrate established risk analysis methods, like HazOp, FTA, and FMEA with UML modeling in a framework for conducting risk assessments of dependable IT-systems.

An important motivation for the metamodel is the practical use of UML to support risk management in general, and risk assessment in particular. In model-based risk assessment, UML models are used for three different purposes:

1. *To describe the target of evaluation at the right level of abstraction*. A proper assessment of technical system documentation is not sufficient; a clear understanding of system usage and its role in the surrounding organization or enterprise is just as important. UML allows these various aspects to be documented in a uniform manner.

2. *To facilitate communication and interaction between different groups of stakeholders involved in a risk assessment*. One major challenge when performing a risk assessment is to establish a common understanding of the target of evaluation, threats, vulnerabilities, and risks among the stakeholders participating in the assessment. This motivates a UML profile aiming to facilitate improved communication during risk assessments, by making the UML diagrams easier to understand for non-experts, and at the same time preserving the well-definedness of UML.

3. *To document risk assessment results and the assumptions on which these results depend to support reuse and maintenance*. Risk assessments are costly and time consuming and should not be initiated from scratch each time we assess a new or modified system. Documenting assessments using UML supports reuse of assessment documentation, both for systems that undergo maintenance and for new systems, if similar systems have been assessed before.

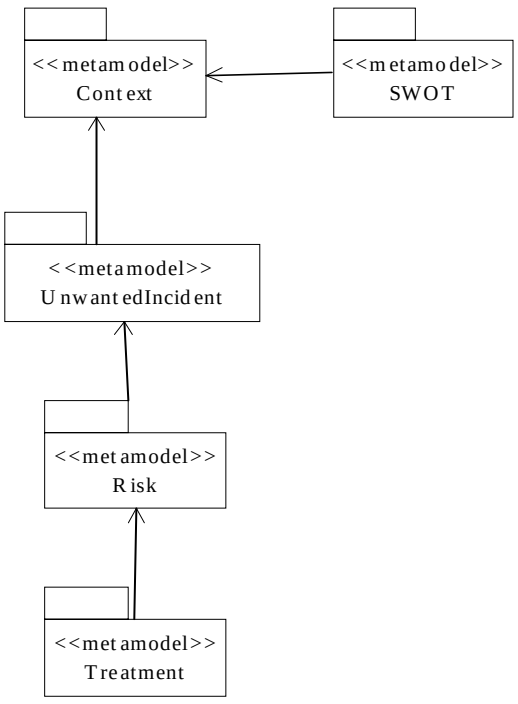

Figure 11.1 - Submodels in the Risk Assessment Metamodel

11.1 Risk Assessment Metamodel

The metamodel is divided into five submodels (Figure 11.1) that support different stages of a risk assessment. A risk assessment always starts with identifying the context of the assessment. A strengths, weaknesses, opportunities, and threats (SWOT) analysis may be part of this. After the context has been established, the remainder of a risk assessment can be divided into identification and documentation of unwanted incidents, risks, and treatments. This process is in accordance with the risk management process of [3].

The unwanted incident model is concerned with organizing and documenting the threats and vulnerabilities that open for incidents that may harm the system. The risk model quantifies unwanted incidents with respect to the reductions of asset value that they may cause. The treatment model supports documenting ways of treating the system and quantifying the effect of treatments with respect to reducing the potential harm of risks.

11.1.1 Context

This submodel (Figure 11.2) defines the context of a risk assessment. The context consists of the stakeholders and assets of the system under assessment, which all further assessment is based on.

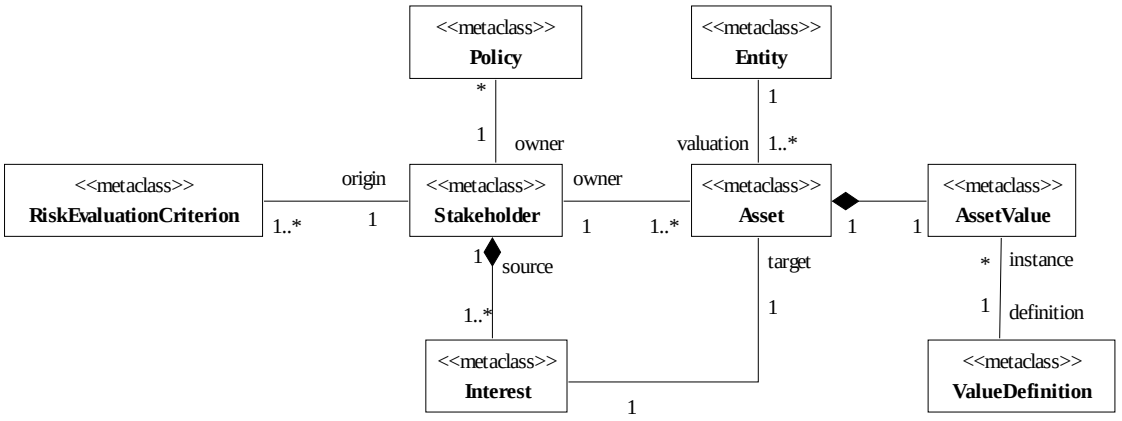

Figure 11.2 - Context submodel

A risk assessment is *asset-driven*, which means that assessment is carried out relative to the identified assets. In the general case, an asset may be anything that stakeholders of the system under assessment find to have value. However, in the setting of Quality-of-Service, an asset should be the *quality level* of an entity of the assessed system. In this case, the entity would typically be a service of the system under assessment.

Each asset may only be related to one stakeholder and should have an unambiguous value assigned by one stakeholder. If two stakeholders view the same entity as an asset, the entity should be documented as two different assets related to the same entity. Two assets are per definition different if valued by different stakeholders. Both the values and the reasons for the valuing may be different.

Below the concepts of the models are described:

Stakeholder: A person or organization who has interests in the assessed system.

Policy: A rule or regulation defined by a stakeholder, related to the system under assessment. A policy could relate to security aspects like confidentiality, integrity, availability, non-repudiation, accountability, authenticity and reliability, and should provide directions for the assessment.

RiskEvaluationCriterion: A criterion that identified risks are evaluated against in order to decide whether the risk is acceptable or not.

Asset: A part or feature of the system that has value for one of the stakeholders, for example the quality level of a service.

Entity: A physical or abstract part or feature of the system under assessment that becomes an asset when assigned value by a stakeholder, for example a service provided by the system.

AssetValue: The value assigned to an asset by a stakeholder.

ValueDefinition: Definition of value types for various values used in a risk assessment, such as asset value.

11.1.2 SWOT

Strengths, weaknesses, opportunities, and threats (SWOT) analysis is a part of establishing the context of a risk assessment. A SWOT is carried out on enterprise level and is used for pointing out general directions of the assessment. Its results are only indirectly used in the further assessment. For this reason the concepts of the submodel for SWOT, shown in Figure 11.3, are not strongly connected to the rest of the metamodel.

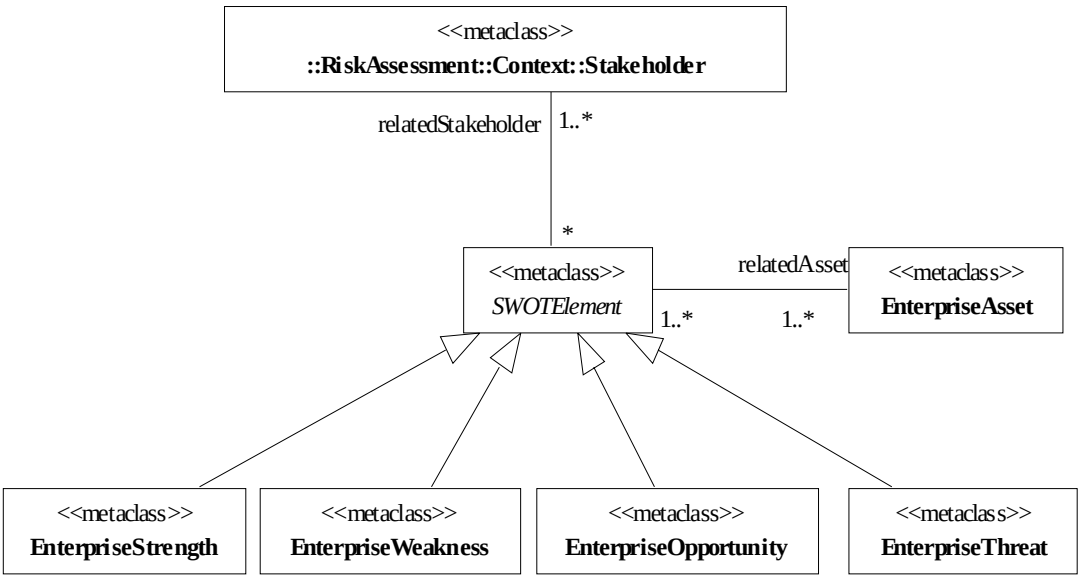

Figure 11.3 - SWOT submodel

A SWOT analysis is concerned with identifying the strategic context of the organization carrying out a risk assessment. The elements of the SWOT model are described below:

EnterpriseAsset: Asset of the organization from a strategic point of view.

EnterpriseStrength: A strategic strength of the organization.

EnterpriseWeakness: A strategic weakness of the organization.

EnterpriseOpportunity: A strategic opportunity of the organization.

EnterpriseThreat: Something that threatens the strategic position of the organization.

11.1.3 Unwanted Incident

Identification and documentation of unwanted incidents is concerned with exploring the threats and vulnerabilities of the system under assessment, and how threats and vulnerabilities may combine and lead to potential incidents that can harm the system.

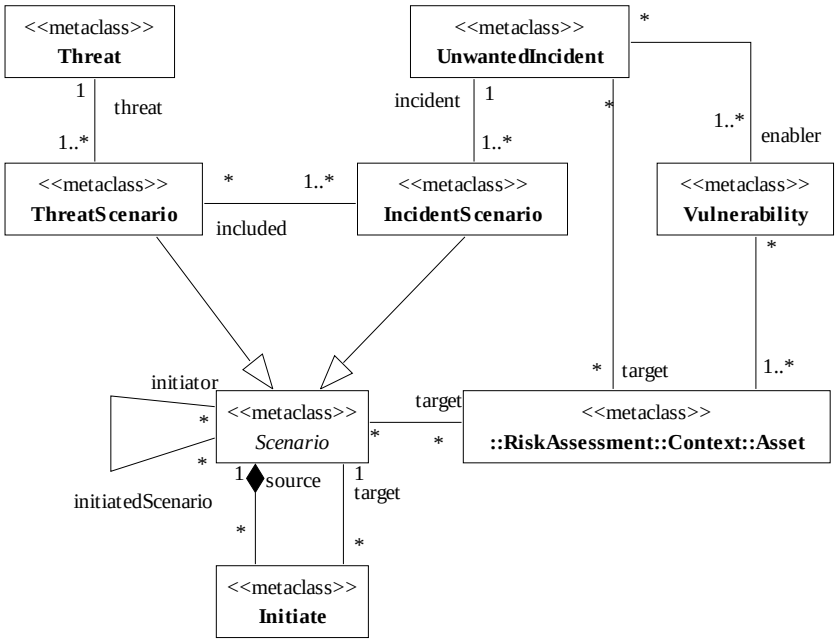

Figure 11.4 - Unwanted incident submodel

The concepts of Figure 11.4 are described below:

Threat. A potential cause of an unwanted incident, which may result in harm to a system or organization and its assets. Threat agents may be external, (e.g., hackers or viruses) or internal (e.g., system failures or disloyal employees).

ThreatScenario. A description of how a threat may lead to an unwanted incident.

Vulnerability. A weakness with respect to an asset or group of assets that can be exploited by one or more threats.

UnwantedIncident. An undesired event that may reduce the value of an asset.

Initiate. An unwanted incident may lead to another scenario. Initiate is a relation for modeling that between an unwanted incident acts as an initiator of another unwanted incident.

IncidentScenario. A scenario leading to an unwanted incident.

11.1.4 Risk

A risk is an unwanted incident that has been assigned consequence and frequency values. These values are used for calculating a risk value, which represents loss of asset value of the asset the risk is related to. Risks that in some way are related or similar may be categorized into risk themes. A risk theme is itself assigned a risk value based on the risks it contains and is treated like a singular risk with respect to evaluation and treatment.

Risk values are evaluated by risk evaluation criteria defined in the context of the risk assessment. A risk evaluation criterion states which risk values are acceptable, and which are not – implying the need for treatment.

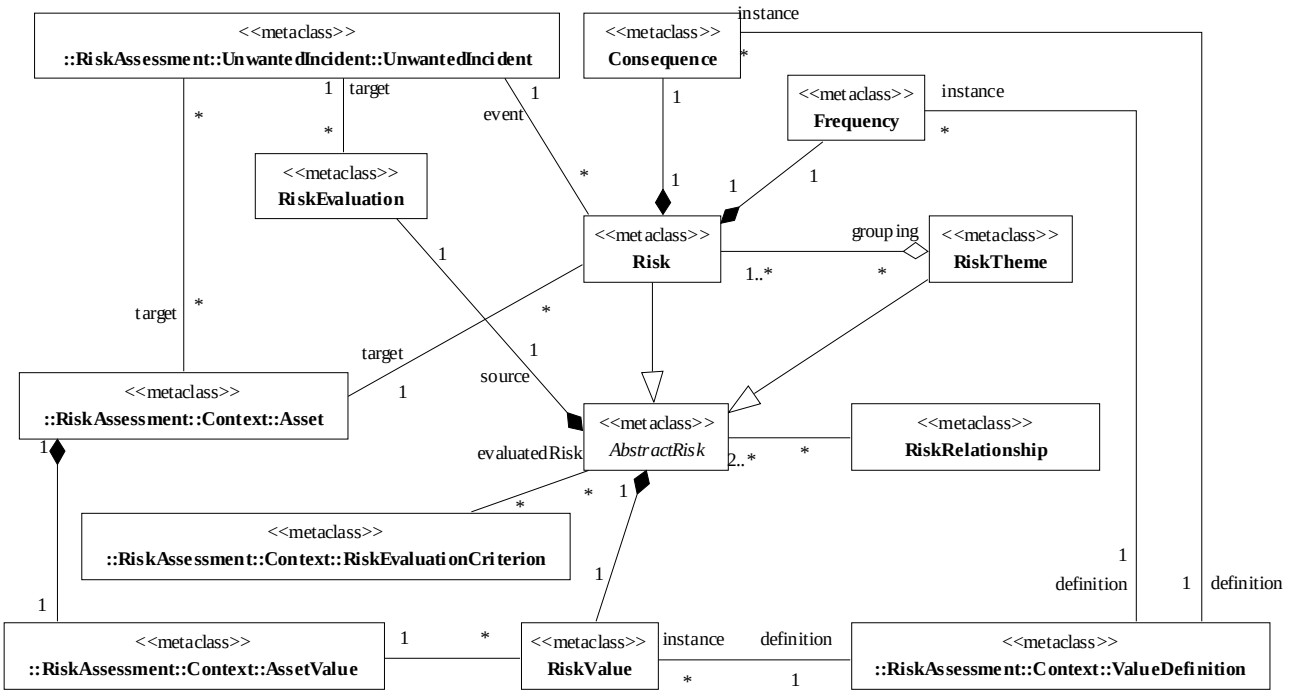

Figure 11.5 - Risk submodel

The concepts of the submodel of Figure 11.5 are described in the following:

AbstractRisk: The common properties of risks and risk themes, such as risk value.

Risk: An unwanted incident that has been assigned a consequence value, a frequency value, and a resulting risk value. Threats, vulnerabilities, and unwanted incidents may go to several assets, but since a risk may reduce the value of an asset, a risk is only related to one particular asset.

RiskTheme: A categorization of similar risks, assigned its own risk value.

RiskRelationship: The relation between risks or risk themes.

RiskEvaluation: The assignment of a risk or a risk theme to the unwanted incident it evaluates with respect to risk value.

Consequence: The consequence of an unwanted incident happening, relative to an asset.

Frequency: A qualitative or quantitative measure of how often or with what probability a risk occurs.

RiskValue: A value assigned to a risk, reflecting the loss of asset value that the risk represents.

11.1.5 Treatment

The treatment model (Figure 11.6) is concerned with documenting and evaluating ways of providing treatments to the system under assessment in order to reduce the value of risks. A treatment may apply to several unwanted incidents. However, when a treatment's capability to reduce risk value is assessed, this is with respect to a single risk or risk theme.

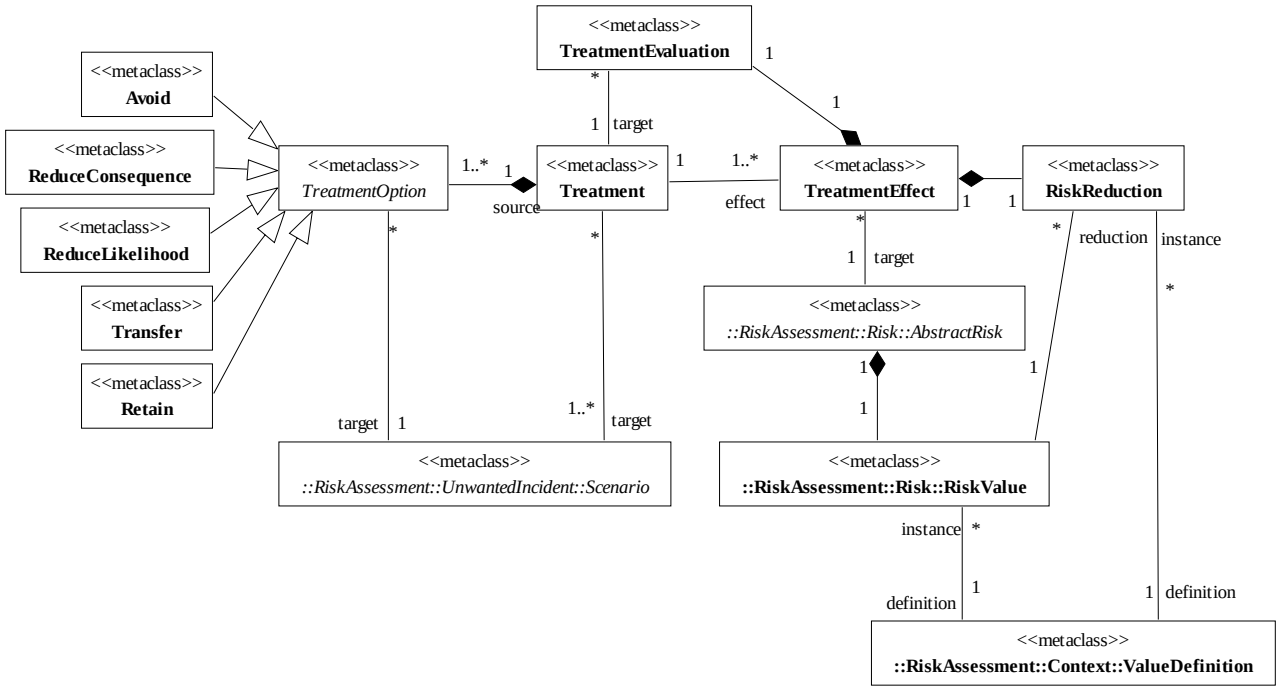

Figure 11.6 - Treatment submodel

The concepts of Figure 11.6 are described below:

Treatment: Ways of treating scenarios leading to risks.

TreatmentEffect: A treatment's capability to reduce the risk value of a particular risk.

TreatmentEvaluation: The assignment of a treatment effect to the treatment it evaluates.

RiskReduction: The value of a treatment effect, i.e., the concrete reduction of a value of a risk.

TreatmentOption: Main classes of providing treatment [3], and hence the relation between a treatment and the scenario it applies to. The options are

Avoid: Decide not to carry on the activity that may lead to risks.

ReduceConsequence: Reduce the impact on assets of the resulting risks.

ReduceLikelihood: Reduce the frequency of the scenario leading to risks.

Transfer: Involve other party bearing or shearing the resulting risks.

Retain: Keep the resulting risks..

11.2 Risk Assessment Profile

This profile provides an extension to the UML metamodel, introducing modeling elements for the concepts defined in the risk assessment metamodel. To some extent, modeling elements from the QoS framework profile in Chapter 8 are used. The structure of the profile, shown in Figure 11.7, reflects the structure of the metamodel.

11.2.1 Context

The subprofile for the context of risk assessments is shown in Figure 11.8. As can be seen from the figure, Stakeholder and Asset may be modeled as both Class and Actor. Documenting assets stakeholder, and their relationships is most appropriately done in a class diagram, and hence assets and stakeholders are modeled as classes. However, when documenting threats and unwanted incidents in use case diagrams (see Section 11.2.3) assets and stakeholders should be modeled as Actor. The Interest relation is modeled using DirectedRelationship.

For the remaining concepts Policy, ValueDefinition, AssetValue, and RiskEvaluationCriterion we apply the appropriate stereotypes from the QoS framework profile. ValueDefinition is modeled as QoSCharacteristic and AssetValue as QoSValue. Policy is modeled as QoSCharacteristic, which has the expressiveness to capture various security aspects (see Chapter 10). RiskEvaluationCriterion is modeled as QoSRequired.

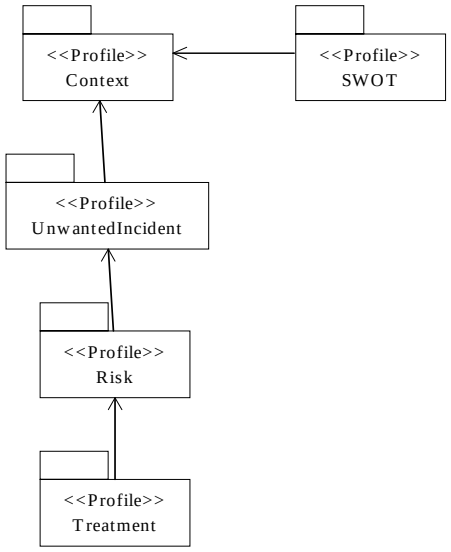

Figure 11.7 - Subprofiles in the Risk Assessment profile

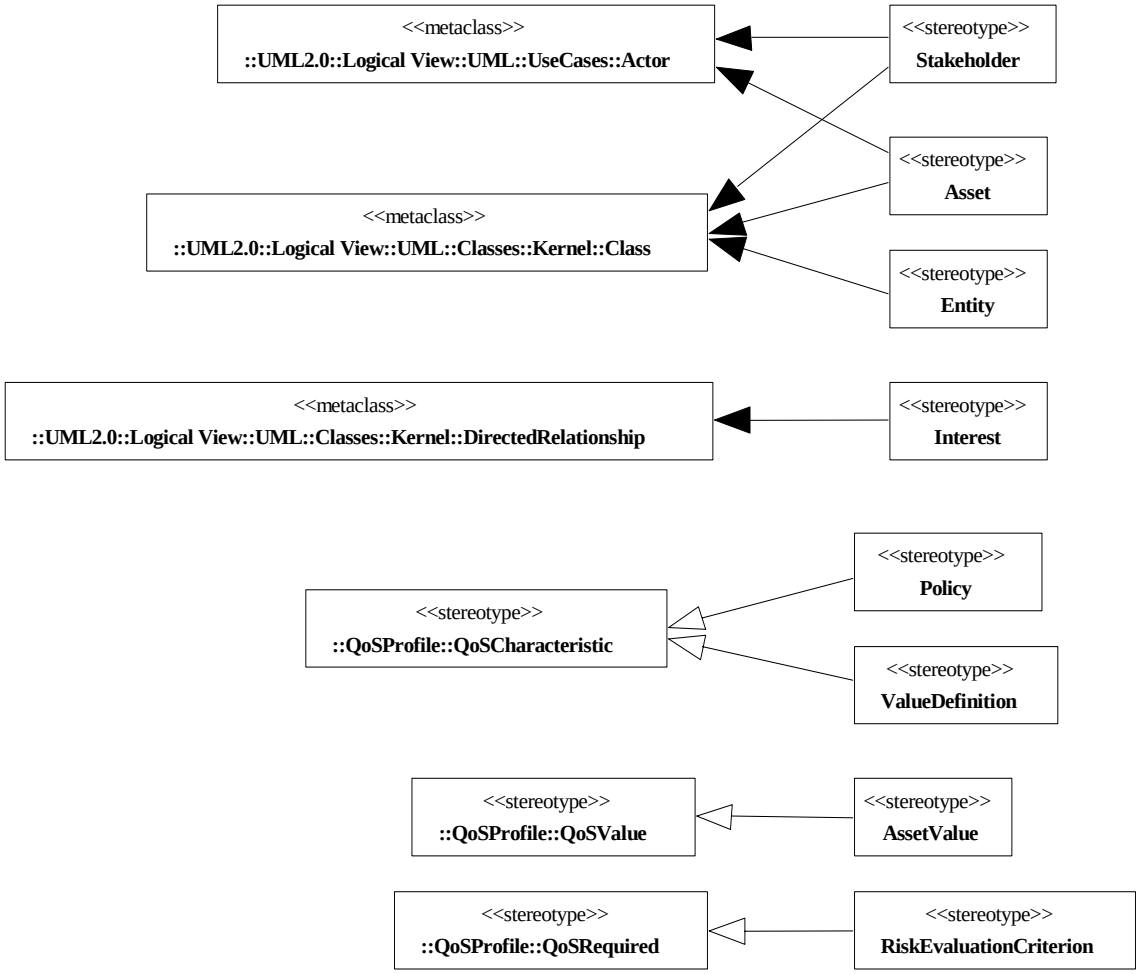

Figure 11.8 - Context subprofile

11.2.2 SWOT

As seen in Figure 11.9, SWOTElement is modeled as EnterpriseAsset as Classifier.

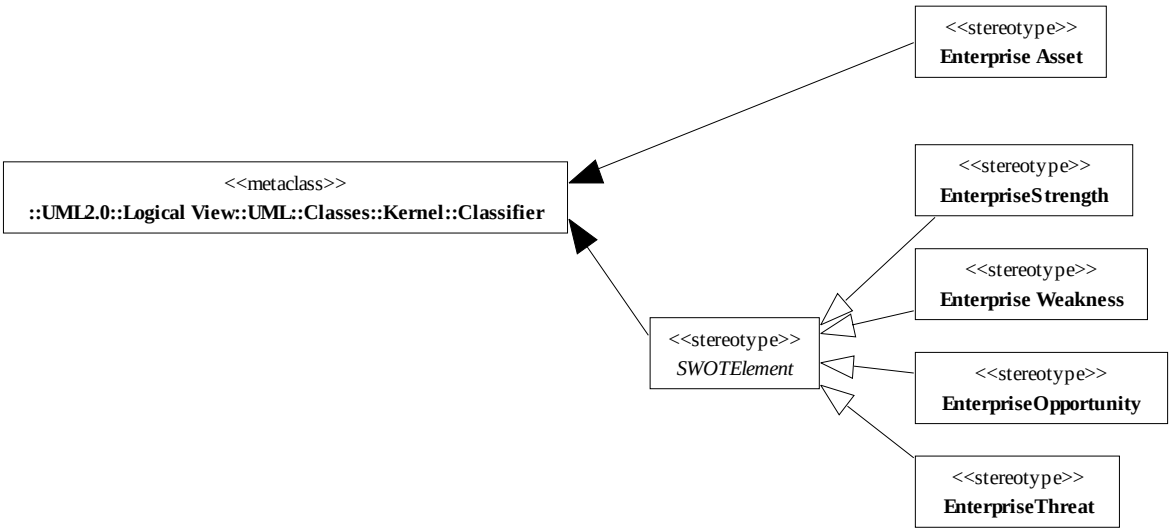

Figure 11.9 - SWOT subprofile

11.2.3 Unwanted Incident

The subprofile for unwanted incidents is shown in Figure 11.10. As the acting part Threat is modeled as Actor and ThreatScenario, as the behavioral aspect, is modeled as UseCase. IncidentScenario, which also represents behavior, is also modeled as UseCase, while UnwantedIncident is not given an explicit representation. Initiate is represented by DirectedRelationship. Vulnerabilities may be seen as (unwanted) features of the assets they apply to, and are modeled as Feature.

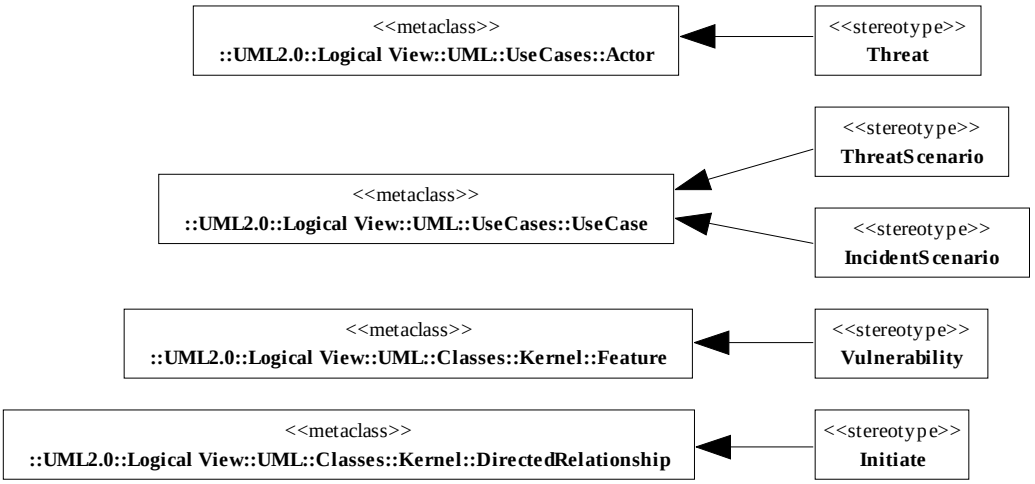

Figure 11.10 - Unwanted Incidents subprofile

11.2.4 Risk

The subprofile for risks is shown in Figure 11.11. Both Risk and RiskTheme are modeled as UseCase. This makes it possible to capture arbitrary grouping of Risks into RiskThemes by making risks parts in a risk theme. RiskRelationship is modeled as Association, allowing risk themes, with their relations, to be documented in class diagrams. RiskEvaluation assigns a risk to an unwanted incident, and is modeled as DirectedRelationship.

Consequence, Frequency, and RiskValue, which all are values, are modeled by the means of QoSValue.

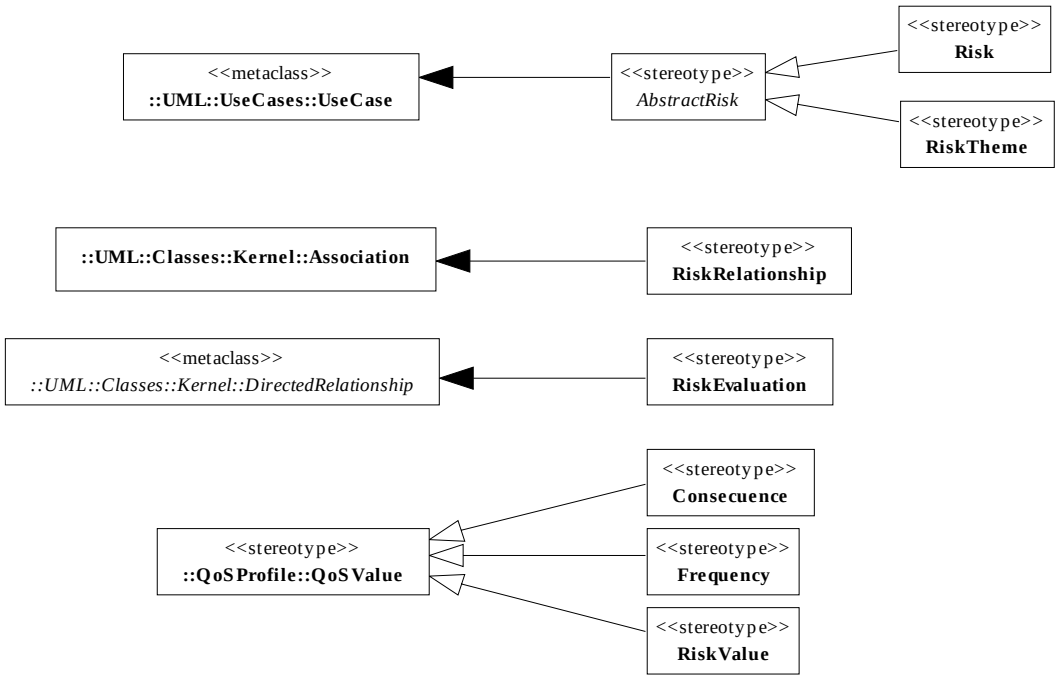

Figure 11.11 - Risk subprofile

11.2.5 Treatment

The treatment subprofile is shown in Figure 11.12. Treatment protects against risks, and is modeled as a UseCase. TreatmentEffect is modeled using Class and TreatmentEvaluation using DirectedRelationship. TreatmentOption relates treatments to risks and is modeled using DirectedRelationship. Finally, RiskReduction is a kind of a value and is modeled with QoSValue.

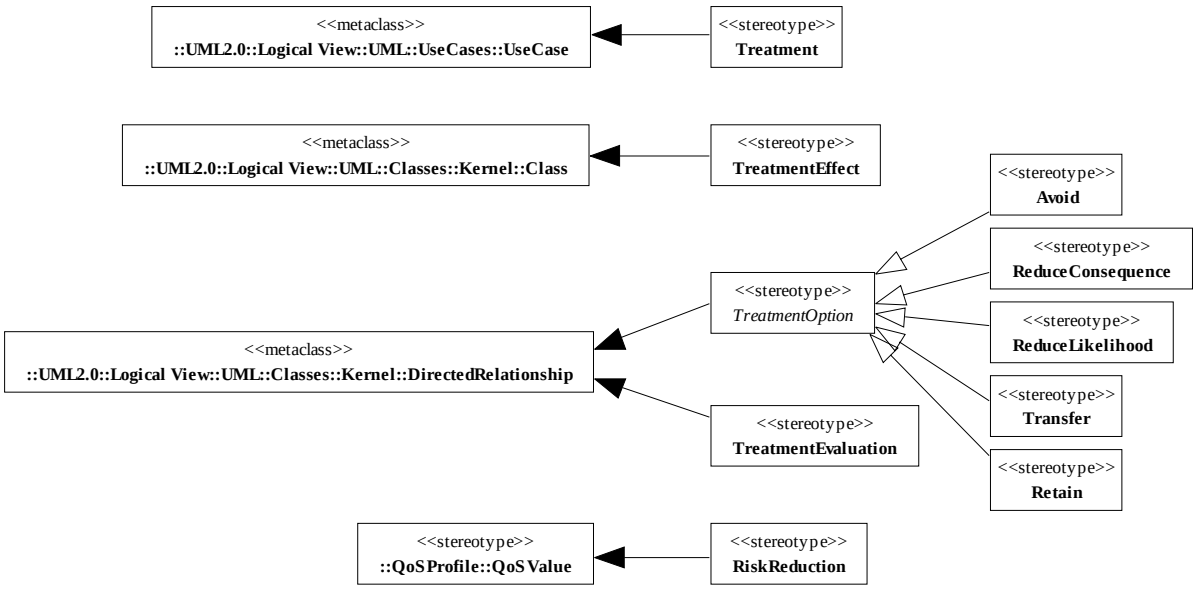

Figure 11.12 - Treatment subprofile

11.3 Examples

In the following we present some examples on the use of the risk assessment profile. The presentation is structured according to the subprofiles.

11.3.1 Context

Figure 11.13 shows how the stereotype <<ValueDefinition>> is used for defining the value types used throughout a risk assessment. In this case all values are enumerations, i.e., values on an ordinal scale, except for “RiskReductionRef” which defines a mapping. An alternative could have been to define asset values and consequences as monetary values and frequencies as probabilities.

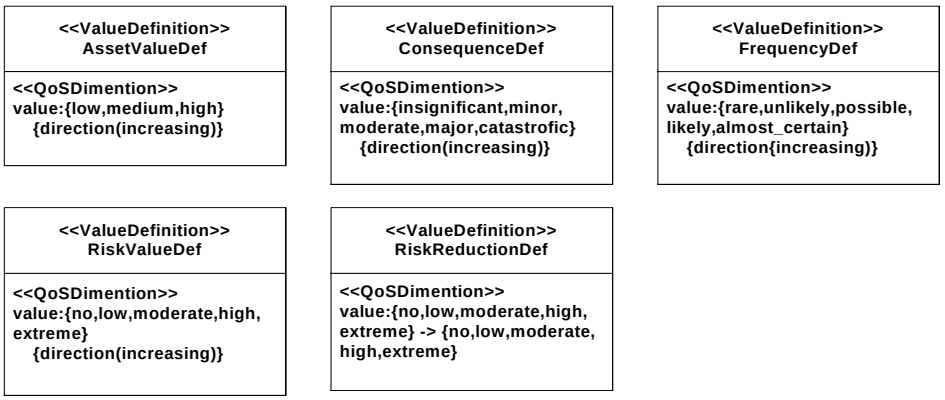

Figure 11.13 - Value definitions

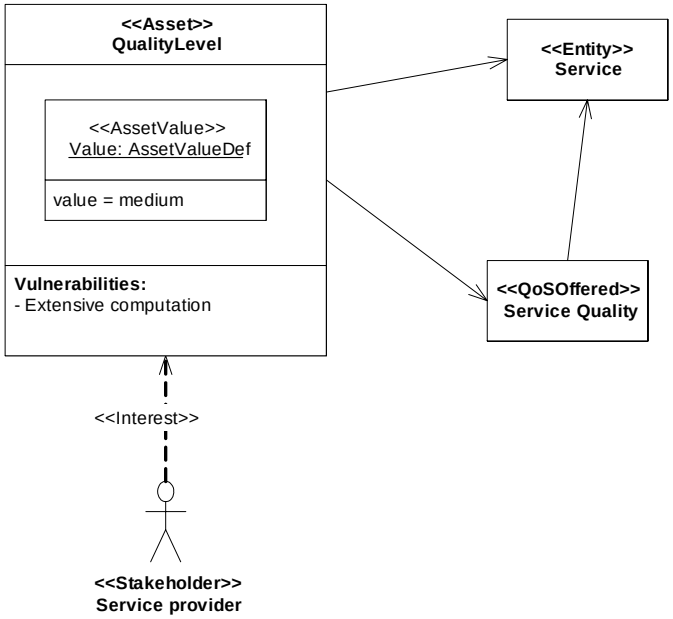

Figure 11.14 - Modeling of assets

Figure 11.14 shows definition of an asset. The entity is a service that has some quality characterizations associated with it. The asset is defined as the quality level of the service related to some offered service quality. The asset is owned by the stakeholder "Service provider," and its value is assigned by instantiating the value definition for asset values. Further the diagram shows that asset has one vulnerability.

11.3.2 Unwanted incident

In Figure 11.15, modeling of a threat is exemplified. The threat "Malicious person" has the scenario, i.e., behavior, "Flooding." This threat scenario is related to the asset "QualityLevel."

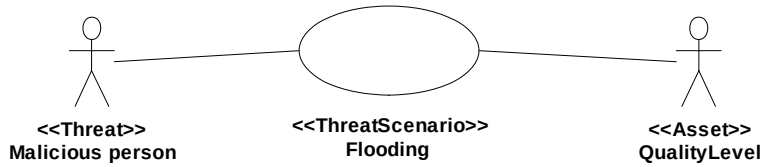

Figure 11.15 - Modeling of threats

Figure 11.16 illustrates how incident scenarios are modeled. The incident scenario "Denial-of-Service" relates to the asset "QualityLevel," and includes the threat scenario from the diagram above. A scenario may lead to another scenario, and this is shown by use of the stereotype <<Initiate>>. In this case, "Denial-of-Service" initiates the incident scenario "Loss of customer" that relates to the assets "Customers."

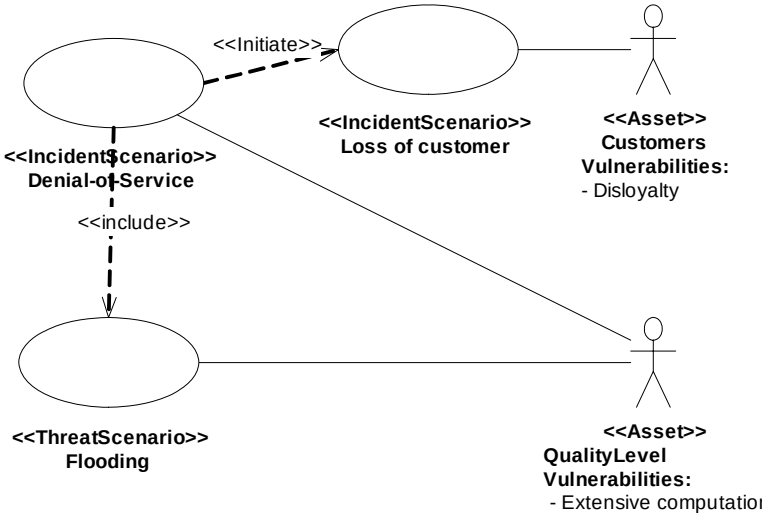

Figure 11.16 - Modeling of unwanted incidents

11.3.3 Risk

A risk is an assignment of consequence, frequency, and risk values to an unwanted incident. Figure 11.17 illustrates how this is modeled. The values are instances of the corresponding value definitions. The risk "Denial-of-service evaluation" is assigned to the unwanted incident resulting from the incident scenario "Denial-of-Service" by the use of the stereotype <<RiskEvaluation>>. The diagram also shows that the risk is related to the asset "QualityLevel."

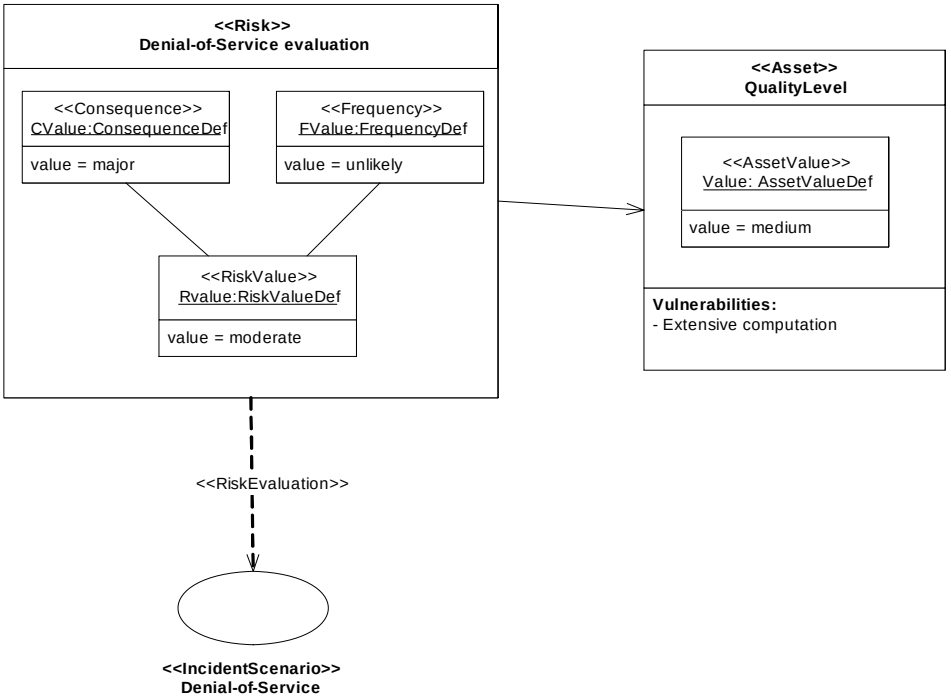

Figure 11.17 - Modeling of risks

Similar risks may be grouped into risk themes. Figure 11.18 shows how the stereotype <<RiskTheme>> is used to define risk themes of instances of risks. This allows a risk to be a member of several risk themes. In this example, the risks "Denial-of-service evaluation" and "Loss of customer evaluation" are grouped to form the risk theme "DoSRelated." As seen in the example, a risk theme is also assigned a risk value.

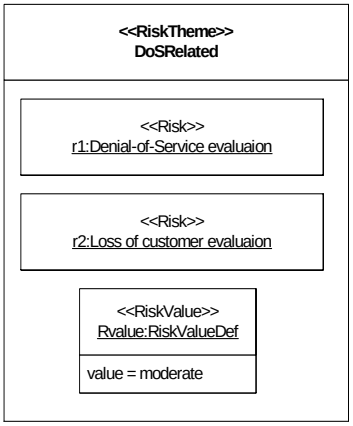

Figure 11.18 - Modeling of risk themes

11.3.4 Treatment

Figure 11.19 models "Authentication" as a treatment for the unwanted incident resulting from the incident scenario "Denial-of-Service." The stereotype <<Transfer>> (one of the treatment options) explains what kind of treatment "Authentication" is.

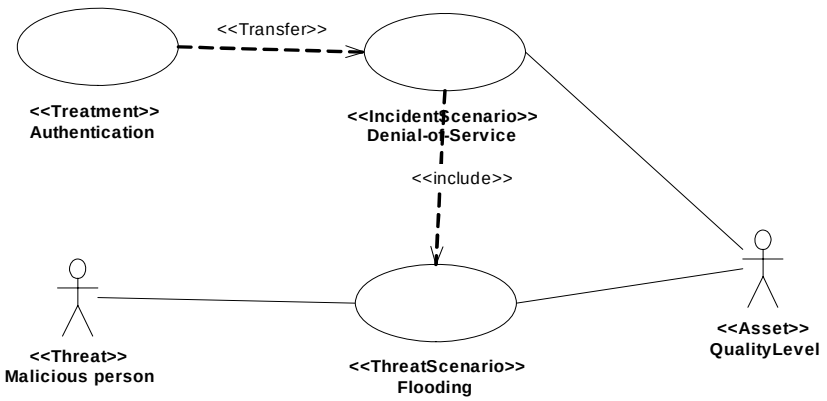

Figure 11.19 - Modeling of treatments

In Figure 11.20 an example of how a treatment effect is modeled is presented. The treatment effect "DoSTransfer" is bound to the treatment "Authentication" by the use of the stereotype <<TreatmentEvaluation>>. The figure also shows that "DoSTransfer" relates to the risk "Denial-of-Service evaluation." The risk reduction, i.e., the value of the treatment effect, is a mapping from moderate to low, meaning that implementation of the treatment would reduce the risk value of "Denial-of-Service evaluation" from moderate to low.

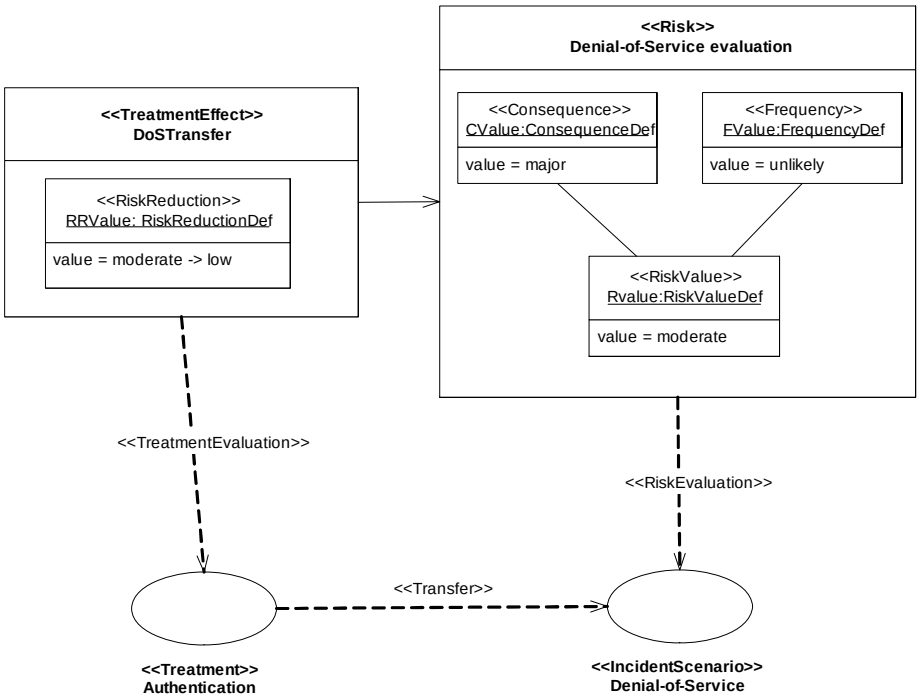

Figure 11.20 - Modeling of treatment effects

12 FT Mitigation Solutions

This chapter suggests some UML extensions for the description of Fault-Tolerant (FT) software architectures. The solutions are oriented to complex systems (in general distributed systems), with high-reliability requirements. Examples of these types of systems are air traffic control systems.

FT techniques provide support to mitigate the reliability problems. To model software architectures that employ FT techniques, some constructors for the description of specific concepts involved in these techniques are required. Some examples are the identification of groups of elements that support services and the selection policies used in multiple versions approaches.

In this specification, the scope of FT to the safety mitigation means is limited. Work was concentrated in the scope of the FT technical solutions, not in safety in general (not in safety assessment and prediction). FT are technical solutions to the reliability requirements. *Reliability* is a specific *QoS Characteristic* that can be quantified in different ways.

Reliability refers to the continuity of the service, and safety is related to the non-occurrence of catastrophic consequences on the environment because of the global system (not only the software system). Reliability measures the probability of failure, not the consequences of those failures. Pressman [22] defines the reliability such as “the probability of failure free operation of a computer program in a specified environment for a specified period of time.”

Software safety is concerned with the consequences of failures from a global system perspective. Leveson [17] defines software system safety as “the software will execute within a system context without contributing to hazards.” A hazard is defined as “a state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object), will lead inevitably to an accident (loss event).”

[8] states “The goal of [software] fault tolerance methods is to include safety features in the software design or Source Code to ensure that the software will respond correctly to input data errors and prevent output and control errors. The need for error prevention or fault tolerance methods is determined by the system requirements and the system safety assessment process.” This is a proposal to integrate the FT solutions in UML software architectures; the safety analysis would be part of a process analysis of *Reliability* QoS.

FT requires some kind of redundancy, fault detection, and recovery. Replication is a basic redundancy tool. In general, replicas will execute in different nodes. UML provide good solutions to describe object-oriented architectures; in our approach, the main redundancy entities will be the objects and components.

This solution provides support to describe *Fault Tolerant CORBA* architectures. However, we are not limited to *Fault Tolerant CORBA* architectures.

12.1 FT Architectures MetaModel

Lyu [18] and Torres [24] propose a classification of FT techniques in two types: i) single-version; in this type of solutions a single piece of software includes some techniques to detect the fault and handle the errors, and ii) multi-version; the same piece of software can have more than one version, and the architecture provides support to avoid global failures when one version fails.

Single and multi-version have different architectures, but both require some basic concepts such as *fault detection*. Some basic concepts of both types of solutions are:

Fault Detection. There are very different types of tools to support the *fault detection*. Some examples are: i) run-time checks, they include hardware detection mechanisms such as overflow, division by zero and others and software checks that raise exceptions. ii) Timing checks, such as watchdog timers. iii) Coding checks based on some kind of redundancy.

iv) Functions and software structures that support some properties such as inverse computations and redundant fields, and v) replication checks based on matching of multiple outputs. The self-protection of FT must take into account the external and the internal contaminations. We can detect errors because of the inputs arriving from other software pieces or errors in the results that are provided to other pieces.

Groups. The replication of software elements requires the identification of the group of elements that compose a replication block to provide the common service. Groups of elements have associated FT policies and styles that customize FT mechanisms according to application characteristics (response time requirements, maintenance considerations, development costs, etc.).

Replication styles. The FT architectures use different policies to handle the different types of replications, and recovery information. Some styles define active replications (all replicas remain active) and others define more passive ones (only one replication is active while others wait for synchronization and wake-up). Some policies require that the state of all replicas be the same, while in others replicas can have divergent behaviors. For passive replications, there are different approaches to update the state of passive replicas. All these types of configuration parameters define the replication styles.

The FT profile includes four metamodels that support the concepts that we have introduced.

- *Fault Tolerant Core:* This package includes the basic concepts for the description of FT architectures. These basic concepts define how to apply FT policies and styles to groups of replications, how to identify these groups, and how to identify the individual replications.
- *Fault Detection:* This package includes solutions for the detection of faults. The different approaches for the detection of faults go from the automatic generation of detectors to the entire support from the application.
- *Object Group Properties:* The group properties include information such as the type of consistency checking, the monitoring of the different members in the group, and how to control the aggregation of new memberships.
- *Replication Styles:* FT techniques use different approaches to support the replications with different properties. The single and multi-version solutions have different replication techniques. Two different styles are passive and active replications that require different types of monitoring and state checking. Another classification depends on the type of information required to synchronize the state of replicas and their persistence.

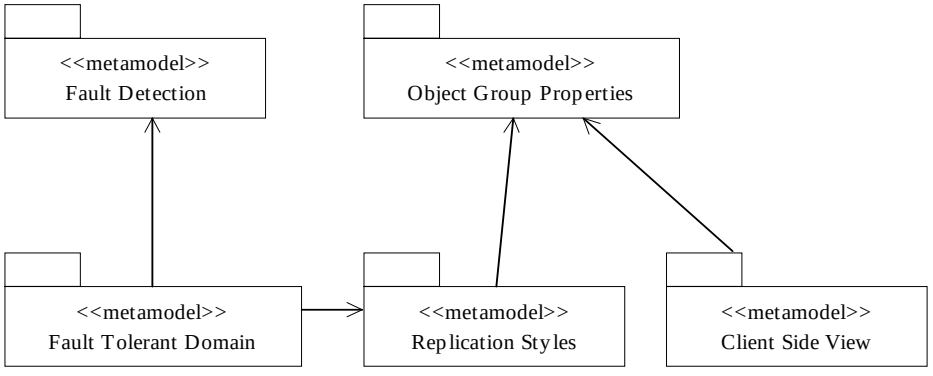

Figure 12.1 - FT Subprofiles

12.1.1 Core Package for FT Architectures

FT extensions support the design of FT systems with the following principles:

- No single point of failure that may cause the loss of a critical function.
- Systematic monitoring and control of applications, nodes, external links and networks to detect failures.
- Manual System Control capability e.g., stop/restart of run-time applications, or transitions to a degraded mode.
- Global and redundant supervision.
- Local supervision capability on each node.
- Widely used synchronization algorithms, and tolerance to loss of external time reference.

The core of FT identifies concepts to model policies that use the common infrastructures that support FT, including the core of group and the core of replica. These main concepts identify the main FT concepts in a UML architecture. These concepts have equivalent elements in CORBA FT (*Fault Tolerant Domain*, *Server Object Group*, and *Fault Tolerant Object*).

Figure 12.2 includes the *core model* for the description of *FT Architectures*. This model includes the following main concepts:

FaultToleranceDomain

Many applications that need fault tolerance are quite large and complex. Managing such applications as a single entity is inappropriate. Each *FaultToleranceDomain* typically contains several hosts and many object groups, and a single host may support several *FaultToleranceDomain*. The *FaultToleranceDomain* decides about the default policies that are applied in the *ServerObjectGroup* and *Replicas* that it manages. The policies includes the approachs to detect the errors and styles of replication management. Each *ServerObjectGroup* has associated a *FaultToleranceDomain*.

Examples of policies are the type of *ReplicationStyle* (e.g., passive, active), initial number replicas, and minimum number replicas. *FaultToleranceDomain* defines the default policies that apply to all object groups associated to this manager. It is also possible to set the properties of an object group.

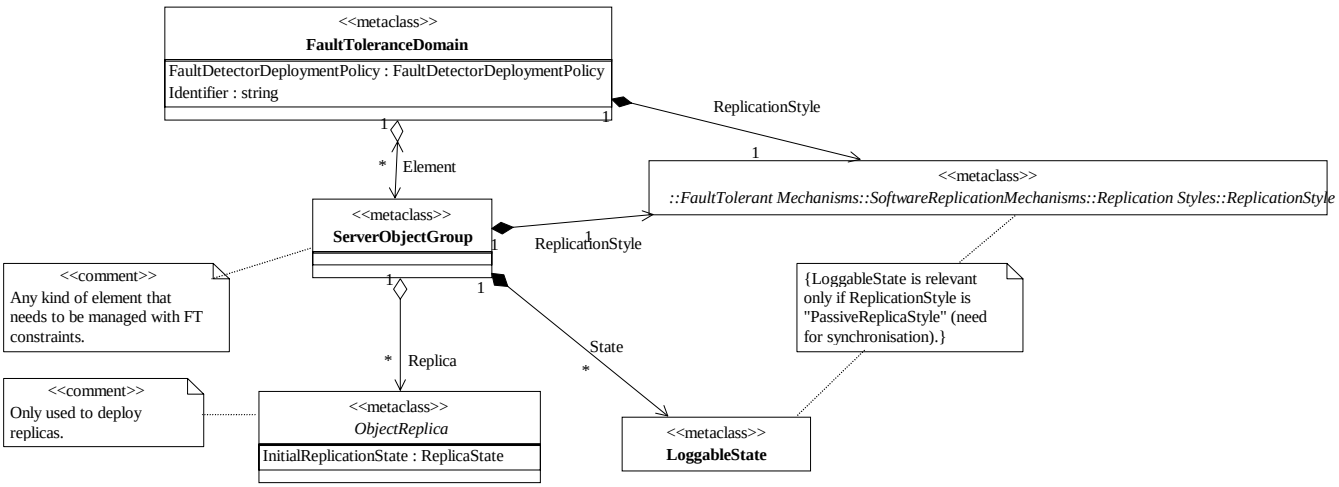

Figure 12.2 - Metamodel of FT Core

ServerObjectGroup

To render an object fault-tolerant, several replicas are created and managed as an object group. While each individual replica has its own identity, a reference to the entire group makes transparent to the clients the concept of replication. The clients invoke the object group, and the *GroupManager* decides the replicas that must execute the invocation and manages the validity of responses.

ObjectReplica

The redundancy entity in this specification is the *Replica*. The number of replicas or their location are basic parameters to support the failure management.

The *LoggableState* defines the significant state of the entire group of *Replicas*. This state is used for the synchronization of primary and backups.

ReplicaState defines the dynamic state information of a *Replica*, which depends on the role of the replica in the group. These roles depend on the policies used in the group but examples are *Primary Replica*, *Backup Replica*, *Transient Replica*. For each type of policy the information included in a *ReplicaState* is different.

The *FaultToleranceDomain* establishes the type of replication and the type of fault detector to be used. Figure 12.3 includes types of fault detection policies.

FaultDetectorDeploymentPolicy

FaultDetectorDeploymentPolicy describes the required material that the safety engineering uses to describe how to monitor software faults. We define three types of detectors:

1. *StaticallyDeployedFaultDetectors*: In an operating environment with a relatively static configuration, location-specific Fault Detectors will typically be created when the FT infrastructure is installed. For example, the stand-alone Fault Detectors could be implemented as daemon processes that are installed with the FT infrastructure. These Fault Detectors could be registered in a manner internal to the FT infrastructure, allowing the infrastructure to include them in every fault-tolerant application within the fault tolerance domain in a transparent manner.

2. *InfrastructureCreatedFaultDetectors*: The FT infrastructure may create instances of Fault Detectors to meet the needs of the applications. Because these Fault Detectors are created (or, at least, configured) by the FT infrastructure, it is the only one who needs to know the identities.

3. *ApplicationCreatedFaultDetectors*: It might be necessary or advantageous for applications to create their own Fault Detectors. For example, applications might have unique knowledge of their operating environment, such as access to hardware indicators of faults within the operating environment. However, unlike the other types of Fault Detectors, the FT infrastructures do not need to know the identity of application-created Fault Detectors.

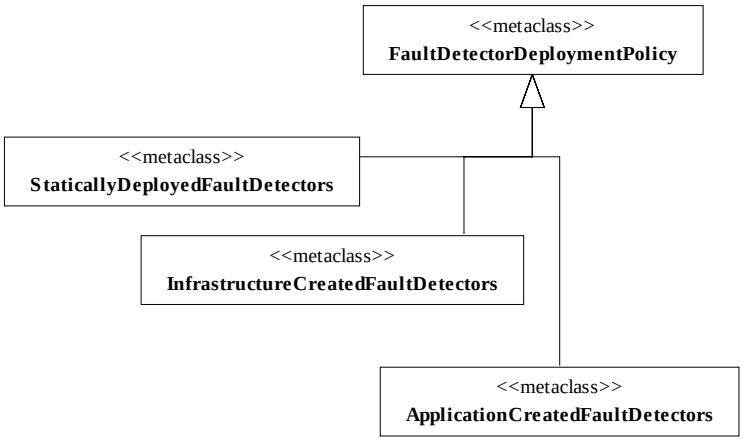

Figure 12.3 - Metamodel of Fault Detection Policies

12.1.2 FT Group Properties

FT infrastructures provide support to detect faults and activate mechanisms to handle these faults. The infrastructures can include different mechanisms to monitor the replicas to detect the failures, to check the consistency, and to handle the faults.

These properties may be set statically as defaults in the *PolicyManager*, or may be set or changed dynamically while the application is executing.

Figure 12.4 includes the metamodel for the description of *FT Group Properties*. This model includes the following concepts:

MembershipStyle

Describes responsibilities for replica creation. Defines whether the membership of an object group is infrastructure-controlled or application-controlled:

ApplicationControlledMembership: The application may create a server object itself and then notify to the *GroupManager* the creation of the new replica. Another alternative is the creation from the *GroupManager* when applications request it. The application is responsible for enforcing the Initial Number Replicas and Minimum Number Replicas properties.

InfrastructureControlledMembership: The *GroupManager* decides when to create the members of the object group, and satisfies the Initial Number Replicas property, and after the loss of a member because of a fault to satisfy the Minimum Number Replicas property. The *GroupManager* initiates monitoring of the members for faults, according to the *FaultMonitoringStyle*.

ConsistencyStyle

Describes responsibilities for replica consistency management. Defines whether the consistency of the states of the members of an object group is infrastructure-controlled or application-controlled. Some components of the FT infrastructure, such as the Logging and Recovery Mechanisms, are used only for object groups that have the infrastructure-controlled Consistency Style.

ApplicationControlledConsistency: The application is responsible for checkpointing, logging, activation and recovery, and for maintaining any kind of consistency appropriate for the application.

InfrastructureControlledConsistency: The FT infrastructure is responsible for checkpointing, logging, activation and recovery, and for maintaining Strong Replica Consistency, Strong Membership Consistency, and Uniqueness of the Primary for the *ColdPassive* and *WarmPassive* Replication Styles.

FaultMonitoringStyle

Describes how replica faults are controlled. Two types of *FaultMonitoringStyles* are:

- *PullMonitoringStyle*: The Fault Monitor interrogates the monitored object periodically to determine whether it is alive.
- *PushMonitoringStyle*: The monitored object periodically reports to the fault monitor to indicate that it is alive.

FaultMonitoringGranularity

The granularity determines the level of control used to detect the fails. Some types require more resources than others, but can detected exceptional occurrences.

IndividualMemberMonitoring: Each individual member of this object group is monitored.

LocationMonitoring: When a new replica in the group is created, and there is not another replication monitored in the same location, the new replica is monitored. This replica acts as a “fault monitoring representative” for the members of the other objects groups at that location. If another object at that location is already being monitored, then that object acts as the “fault monitoring representative” for the member of this object group at that location. If the “fault monitoring representative” at a particular location ceases to exist due to a fault, then the Replication Manager regards all objects at that location to have failed and performs recovery for all objects at that location. If the “fault monitoring representative” ceases to exist because the replica was removed from the group but had not actually failed, then the Replication Manager selects another object at that location as the “fault monitoring representative.”

LocationAndTypeMonitoring: When a new replica of a group is created at a particular location, and no other replica of the same group at that location is already being monitored, then the new replica of this object group at that location is monitored. This member acts as a “fault monitoring representative” for the members of the other object groups of the same type at that location.

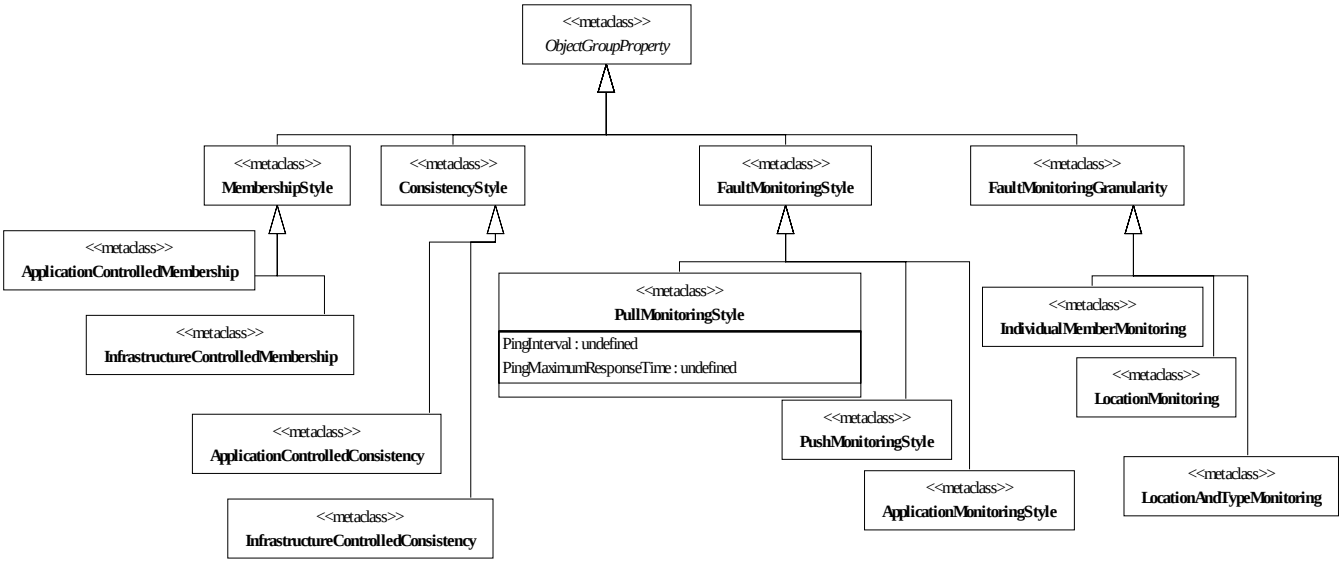

Figure 12.4 - Metamodel of FT Group Properties

12.1.3 FT Replication Styles

FT depends on entity redundancy, fault detection, and recovery. Replicated objects can invoke the methods of other replicated objects without regard to the physical location of those objects. Support for redundancy in time is provided by allowing clients to repeat requests on the server replicas, using the same or alternative transport paths. The re-invocation is transparent to the client.

Figure 12.5 includes the metamodel of *FT Replication Styles*. This model includes the following main concepts:

TransientStateReplicationStyle

This replication style family defines styles for objects that do not have any persistent state.

StatelessReplicationStyle is a type of *TransientStateReplicationStyle*. For the *StatelessReplicationStyle*, the behavior of the object group is unaffected by its history of invocations. A typical example is a server that provides read-only access to a database.

PersistentStateReplicationStyle

This replication style family defines styles for objects that have a persistent state. The infrastructure uses persistent state to reestablish some state.

PassiveReplicationStyle

This replication style family defines replication styles based on the uniqueness of the object replica that is responsible for managing incoming requests (this replica is usually called master or primary).

The *PassiveReplicationStyles* require that, during fault-free operation, only one member of the object group, the primary member, executes the methods invoked on the group. Periodically if infrastructure is controlled, or on demand (if application controlled), the state of the primary member is recorded in a log, together with the sequence of method

invocations. In the presence of a fault, a backup member is promoted to be the new primary member of the group. The state of the new primary is restored to the state of the old primary by reloading its state from the log, followed by reapplying request messages recorded in the log. Passive replication is useful when the cost of executing a method invocation is larger than the cost of transferring a state, and the time for recovery after a fault is not constrained. Two types of *PassiveReplicationStyles* are:

- *WarmPassiveReplicationStyle*: A form of passive replication in which only the primary member executes the methods invoked on the object group by the client objects. Several other members operate as backups. The backups do not execute the methods invoked on the object group; rather, the state of the primary is transferred to the backups periodically.
- *ColdPassiveReplicationStyle*: A form of passive replication in which only one replica, the primary replica, in the object group executes the methods invoked on the object. The state of the primary replica is extracted from the log and is loaded into the backup replica when needed for recovery.

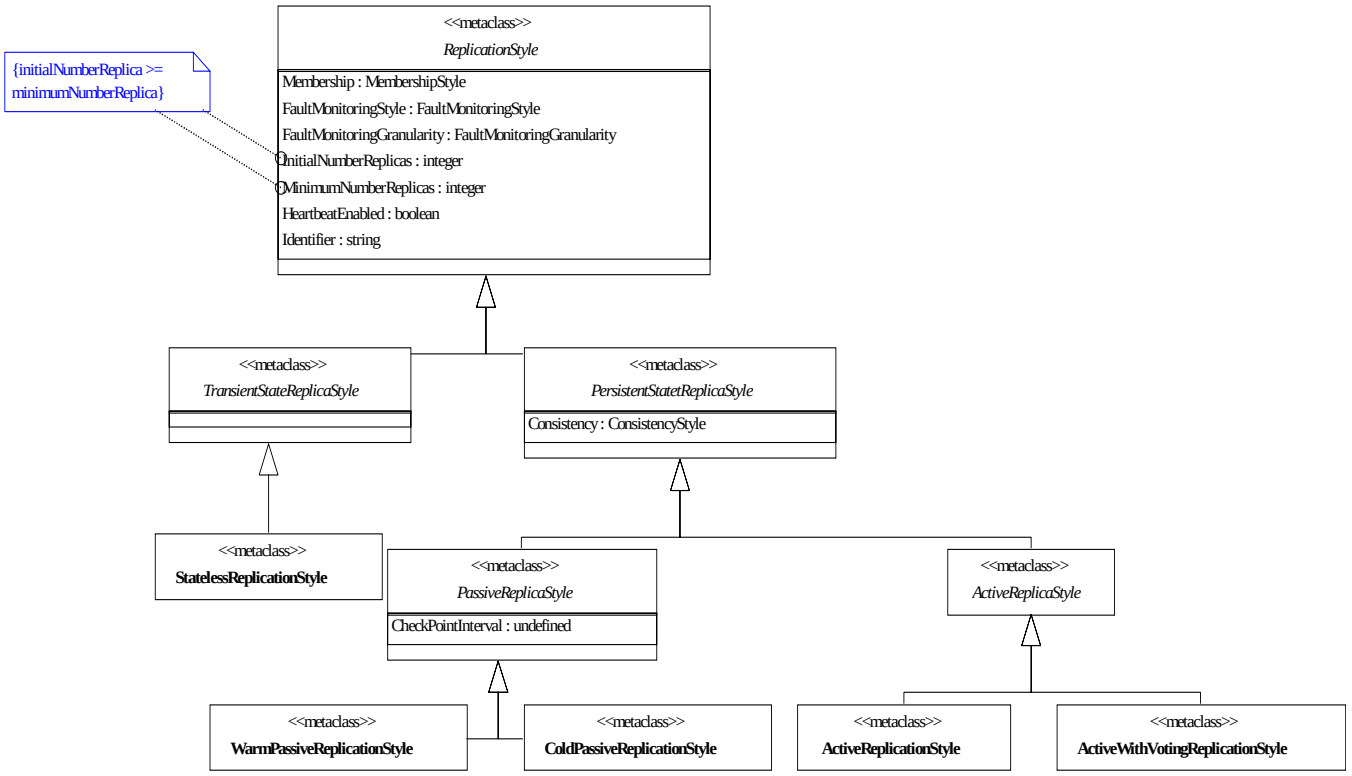

Figure 12.5 - Metamodel of FT Replication Styles

ActiveReplicaStyle

This replication style family defines styles where several replicas of a same object are active simultaneously (e.g., they all compute incoming requests).

The *ActiveReplicationStyle* requires that Figure 9.6 describes the correspondence between the description of template in UML 2.0 and the parameters of *QoS Characteristics*. Figure 9.6 (a) includes a template that represent the parameters description for a QoSCharacteristic and one binding from another characteristic. Figure 9.6 (b) represents the characteristics equivalent in QoS metamodel that creates the binding references between characteristic and associate the

parameters.ll of the members of an object group execute each invocation independently but in the same order. They maintain exactly the same state and, when a fault in one member occurs, the application can continue with results from another member without waiting for fault detection and recovery. Even though each of the members of the object group generates each request and each reply, the Message Handling Mechanism detects and suppresses duplicate requests and replies, and delivers a single request or reply to the destination object(s).

Active replication is useful when the cost of transferring a state is larger than the cost of executing a method invocation, or when the time available for recovery after a fault is tightly constrained. Two types of ActiveReplicationStyle are:

- *ActiveReplicationStyle*: All of the members of an object group independently execute the methods invoked on the object. If a fault prevents one replica from operating correctly, the other replicas will produce the required results without the delay incurred by recovery.
- *ActiveWithVotingReplicationStyle*: They are active replication where the requests (replies) from the members of a client (server) object group are voted, and are delivered to the members of the server (client) object group only if a majority of the requests (replies) are identical.

12.2 FT Architectures Profile

The packages of FT profile include stereotypes from the description of four main concepts: the FT policies for the domains (*FTFaultToleranceDomain)*, the identification of groups (*FTServerObjectGroup*), the state to be considered in state full replicas (*FTLoggableState*, and *FTHasReplicationState*) and the replicas styles (*FTReplicationStyle* and subclasses). Figure 12.6 and Figure 12.7 include the general stereotypes and stereotypes for the description of replication styles. The stereotypes are based on metaclasses and attributes presented in Section 12.1, “FT Architectures MetaModel,” on page 63.

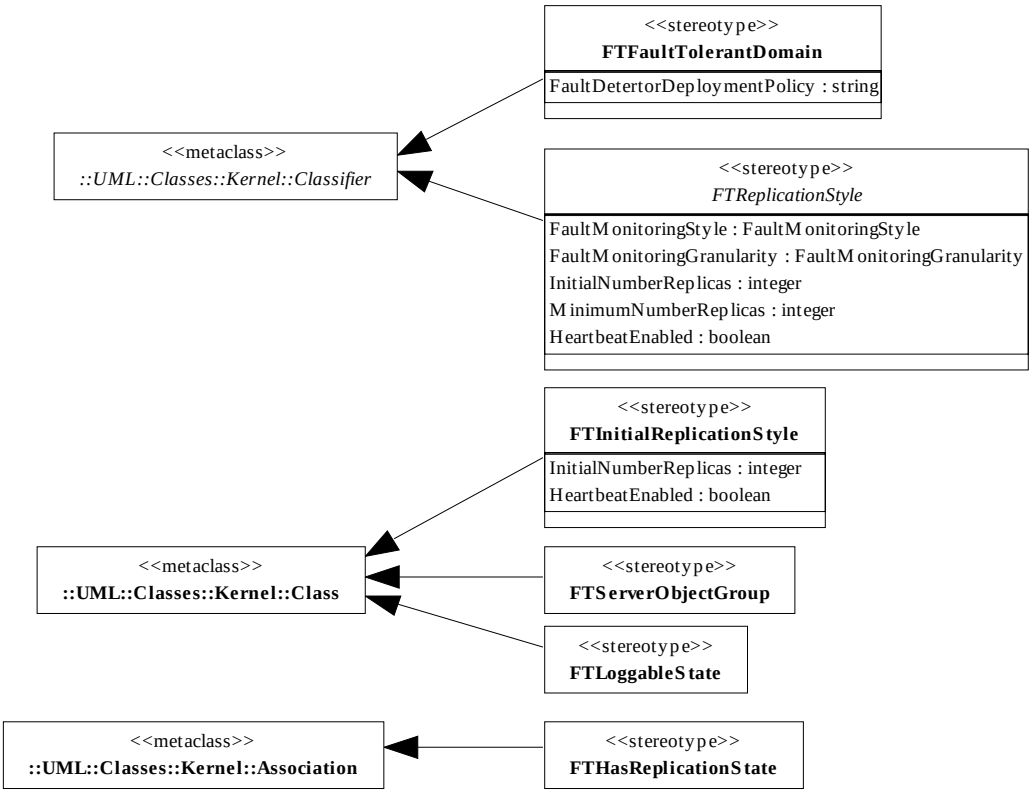

Figure 12.6 - Core FT Profile

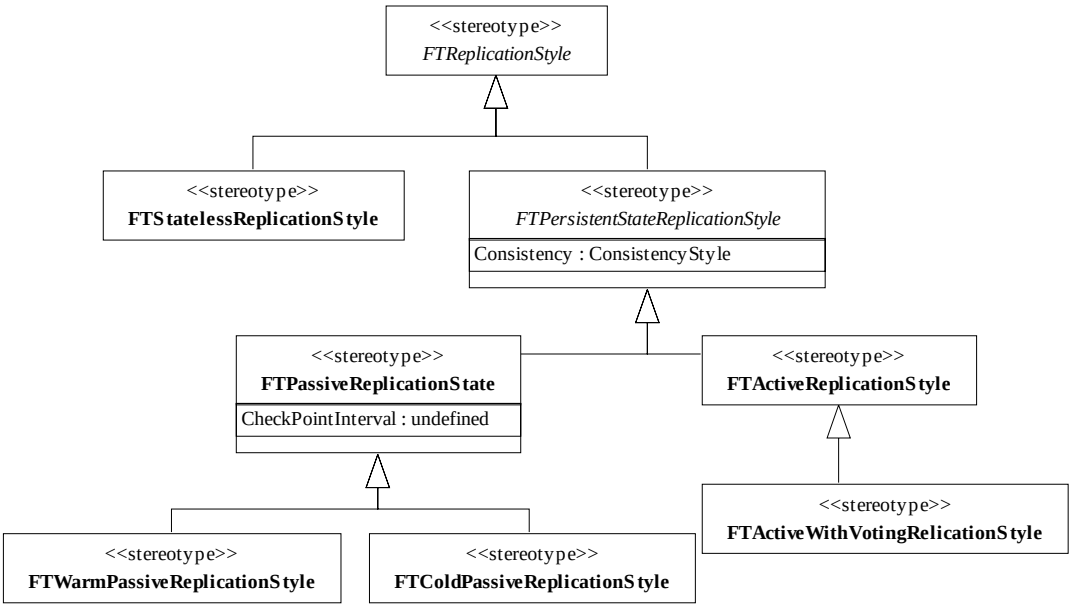

Figure 12.7 - Profile of FT Replication Styles

Annex A: SPT QoS and Resources Conceptual Models

(informative)

This annex includes the models of SPT [20] that we reuse for the description of resource usage with quality annotations. Metamodels included in Chapter 8 do not include metaclasses for the description of quality of resource services. Chapter 4 of SPT pay special attention to this concept.

A model that describes a set of resources, the clients that use these resources, and the qualities that these resources provide must include model elements that identify the resources, the clients, the connection that describes the usage of resources from clients, and the qualities associated to these usages.

Figure A-1 includes the concepts of resource and resource services. They have associated a set of QoS characteristics that qualify the service that provide these resources. This model does not distinguish between the characteristics and the constraint that must fulfill the resource in the services provided.

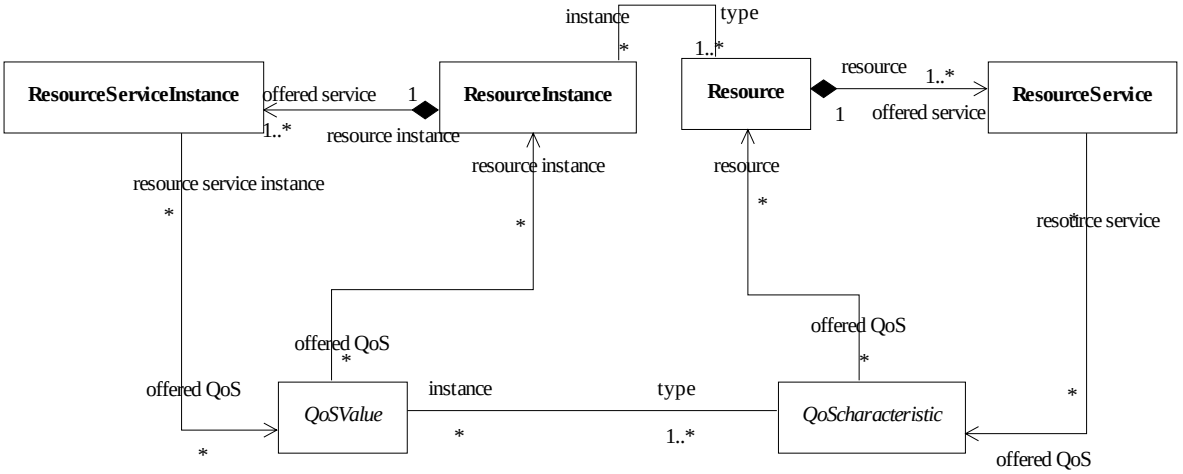

Figure A.1 - General Resource Core

Figure A-2 includes the classes for the identification and description of usage of resources.

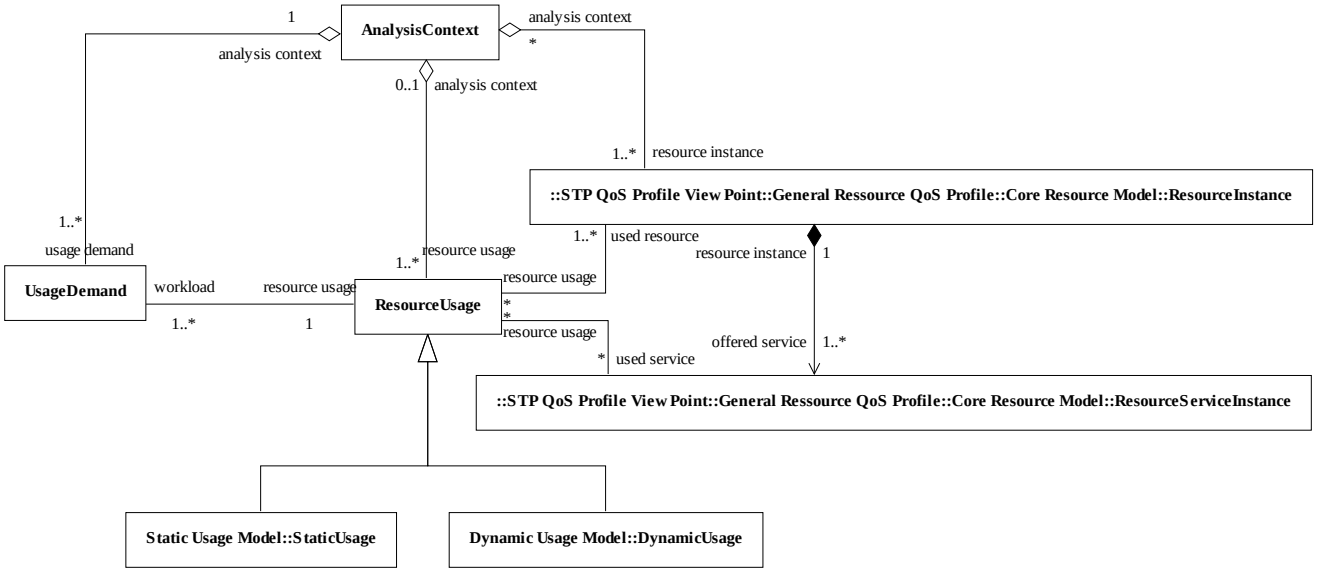

Figure A.2 - Resource Usage Framework

Figure A.3 and Figure A.4 include details for the description of dynamic and static resource usage.

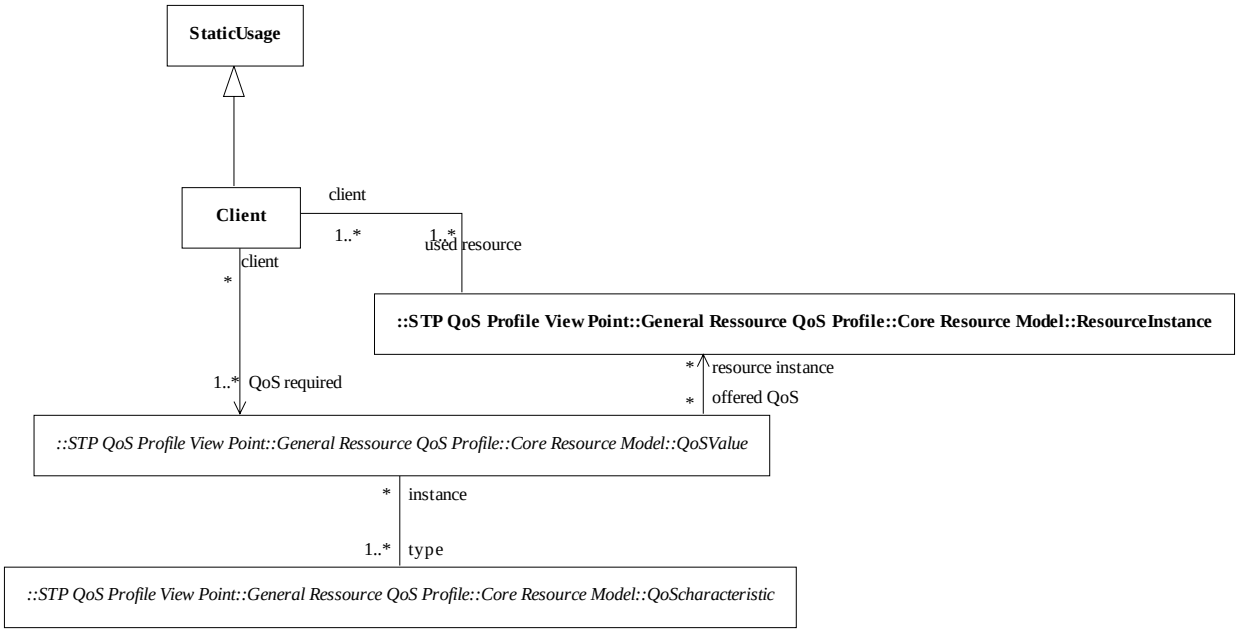

Figure A.3 - Static Resource Usage

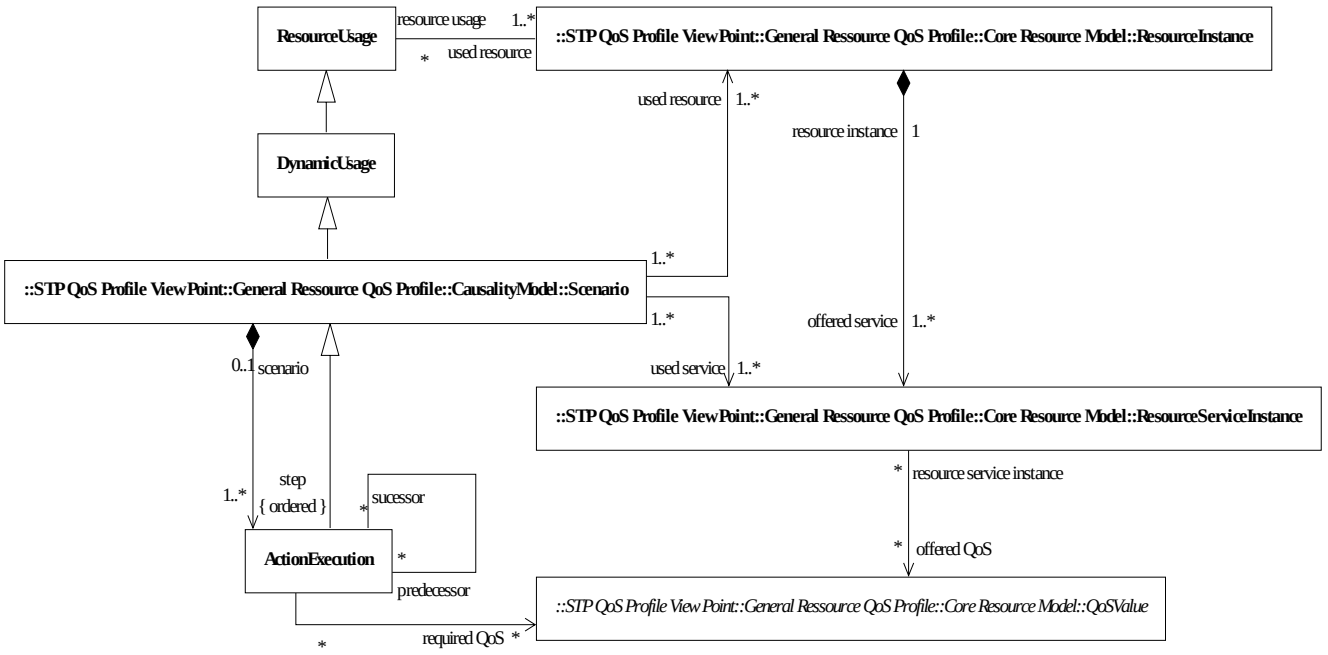

Figure A.4 - Dynamic Resource Usage

Annex B: Proof of Concepts

(informative)

This annex includes guides for the application of QoS extensions, for the description of real-time models analyzable with scheduling analysis techniques. Rate monotonic analysis techniques and scheduling analysis techniques do the analysis of systems based on some analytical models that include some quality characteristics and attributes. These characteristics are basically characteristics for the description of workloads, worst-case latencies, and QoS policies.

We have defined a set of QoS characteristics based on the QoS Catalog included in Chapter 10. The QoS Catalog's design was based on general QoS characteristics included in some standards and quality evaluation references. In this section we have reused the extensions and domain models included in [20] and some modeling languages for the design of analyzable models [10].

Our extensions pay special attention to some modeling aspects of hard real-time techniques not considered in UML:

- *Systems load and load timing distribution*. The performance parameter evaluation depends on the number and type of jobs included in the system, and their space distribution.
- *Resources available and usage of resources*. Different architectural solutions can include different system resources, and can make different resource usage.
- *Scheduling policies*. The performance analysis depends on the type of scheduling algorithms used in the resource management.
- *Representation of analysis results*. The analysis of the models can provide results useful during the architecture evaluation.

B.1 Scheduling Analysis Based on QoS Characteristics

The QoS characteristic `Performance::Demand::demand` included in the QoS Catalog provides support for the description of workloads. But it was not designed for scheduling analysis and does not include some common results of scheduling analysis such as end-to-end response time, or if a trigger is schedulable or not. Figure B.1 includes the characteristic `QoS4SADemand` that reuses the characteristic `demand` (Figure 10.5). In this characteristic the association with the characteristic `arrivalPattern` is especially important, that includes as dimensions values that represent the same concepts as the tagged type RTarrivalPattern in [20].

QoSRequired and *QoSContract* constraint based on `QoS4SADemand` can annotate UML 2.0 *Messages*, *Control Flows*, *Associations*, and *Transitions* for the description of frequencies of demand of services. These UML elements have associated implicitly or explicitly request of services from a client to a service provider, and the *QoS Constraints* provide additional information for the temporal distribution of the invocations. `QoS4SADemand` include some dimensions that are the results of scheduling analysis (`endToEndTime` and `isSchedulable`). A *QoSValue* instance of `QoS4SADemand` can represent these types of results.

Figure B.2 includes two new types of latencies and reuses the characteristic `Performance::Latency::latency` included in the catalog. The characteristic `QoS4SAGlobalLatency` extends the characteristic `latency` that represents the latencies of transactions and group of actions associated to demands. The new latencies allow representing

individual latencies of actions and their laxity. The characteristic `QoS4SADetailedLatency` decomposes the total latency into specific time values such as preemption and blocking times. These dimensions represent some of the concepts included in the class `SAAction` in [20].

The latencies represent the deadlines and output jitters. These values can be associated to model elements (*Messages*, *Control Flows*, *Associations*, *Transitions*, *Actions*, and *Structural Features*) with *QoS Constraints*. These constraints represent the temporal constraints that must achieve the model elements.

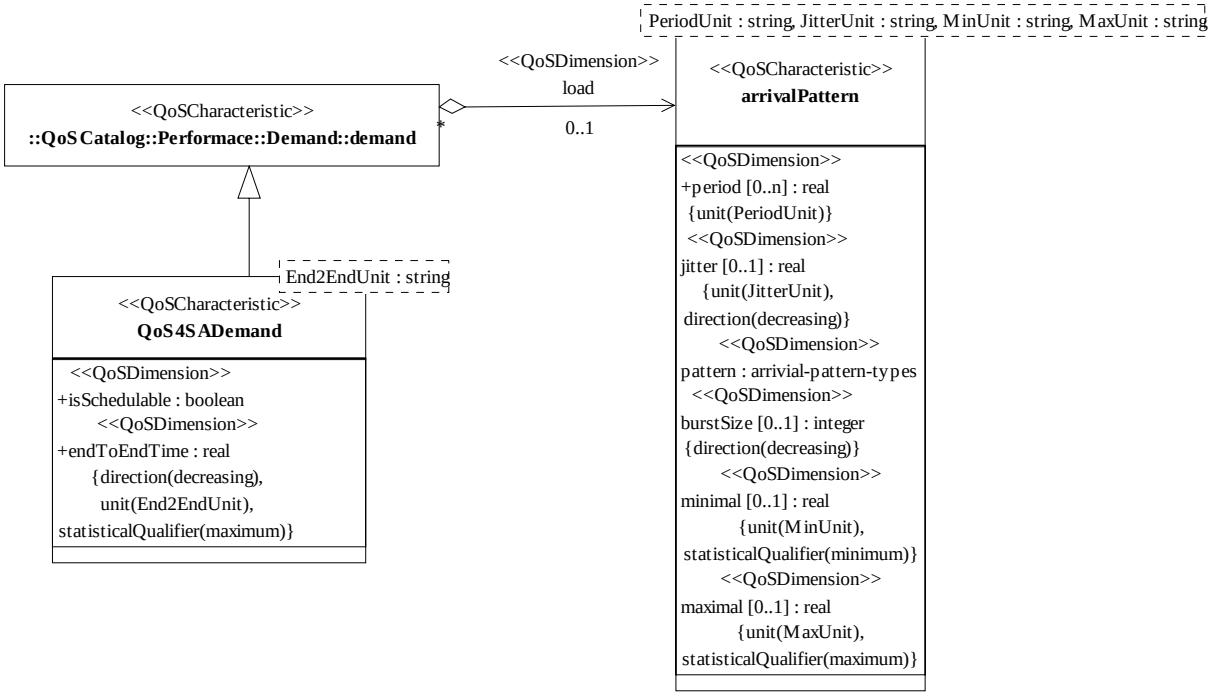

Figure B.1 - Demand Characteristic for Scheduling Analysis

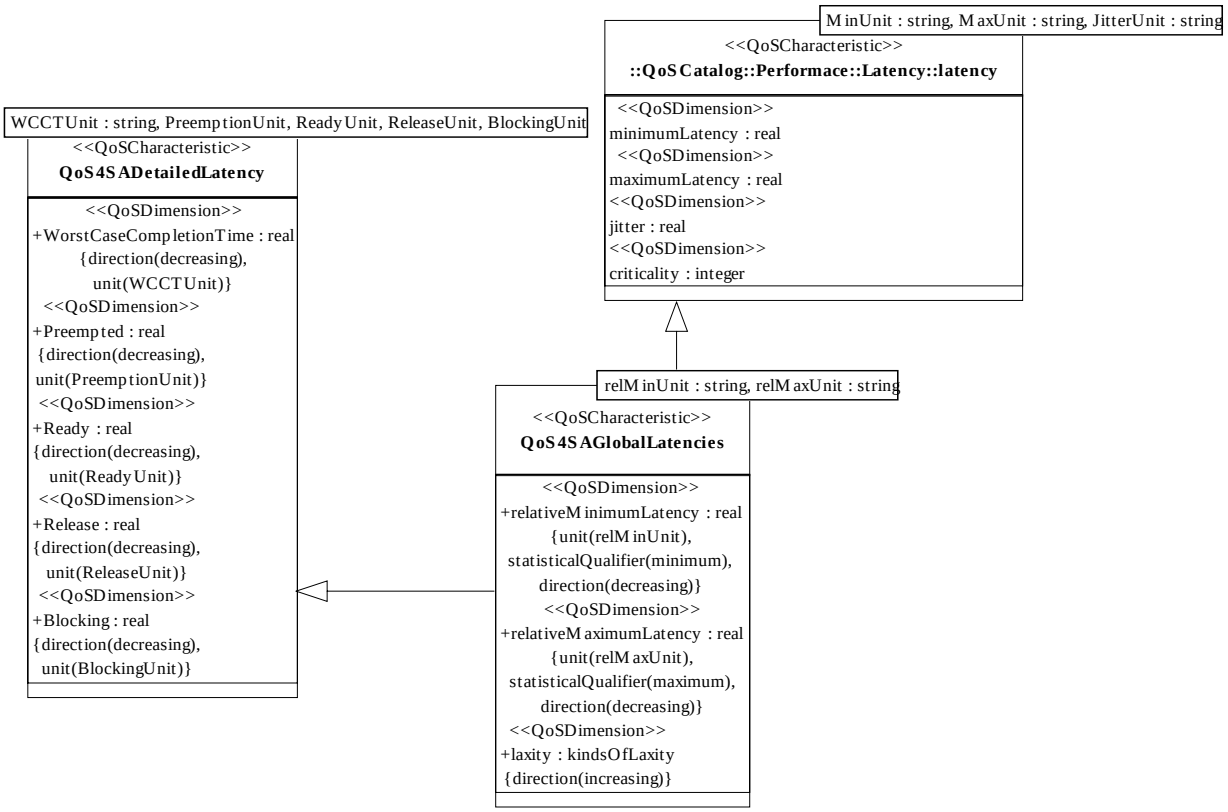

Figure B.2 - Latency Characteristic for Scheduling Analysis

Figure B.3 includes a new characteristic (`QoS4SAResourceUtilization`) that extends `resource-utilization`. Both represent characteristics that are used to describe the quality that resources provide. `QoS4SAResourceUtilization` represents the request of resources that do the actions and transaction in the system. `QoS4SAResourceUtilization` extends `resource-utilization` to specify when a request is atomic.

UML 2.0 model elements that can represent the request of resource services are *Messages*, *Control Flows*, *Associations*, *Transitions*, *Actions* and *Structural Features*. There are two different approached to represent the request of the resource service: i) *Implicitly.* The instance or classifier that include the model element (method of the message, the actions associated to the control flows, the association end of the association, or the transition, action and structural feature) has a *GRMdeploys* relation with the resource, and implicitly the execution of these features uses implicitly the resource. ii) *Explicitly.* Some type of extension annotate the model element to identify the resources that they use. [20] uses both solutions. We are going to use only the second solution to avoid the confusion created because in these cases, in general, there are two services: the service of the resource, and the application service (for example, the operation of the message and the resource used for the execution of the message). The request of the resource service is represented with a *GRMrequires* dependency. And the *QoS Constraints* annotate these dependencies to describe the quality of resource service required and offered.

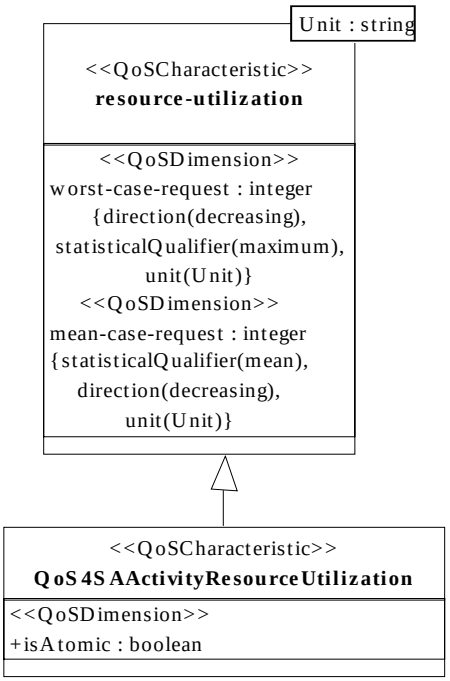

Figure B.3 - Efficiency Characteristic for Scheduling Analysis

In the scheduling analysis techniques are especially important the scheduling policies used for resource management. The QoS Catalog does not pay special attention to this subject, because, in general, these policies are specific of the techniques (e.g., QoS IP and networks, real-time), and most of the middlewares and operating systems of general purposes do not include these policies.

Two new characteristics include dimension for the description of data resources and execution resources. These dimensions are, for example, the context switch times and specific parameters of some scheduling algorithms (e.g., the resource ceiling of ceiling protocols).

These dimensions have the same semantics as the equivalent attributes of classes `ExecutionEngine` and `Sresource` in [20].

Most of QoS characteristics are templates whose parameters must be resolved before being used. The parameters represent, in general, the units of times used in the time expressions. Here, we will use the same units as in [20] (ns, us, ms, s, hr, days, wks, mos, and yrs). After the parameters resolution, we can use these characteristics in the description of *QoSValues* and *QoSConstraints* that represent the specific values of these characteristics. The next section includes some examples of application.

QoS Offered constraint annotates GRMResource model elements (elements with stereotype GRMResource). These constraints define the QoS policies for the resource.

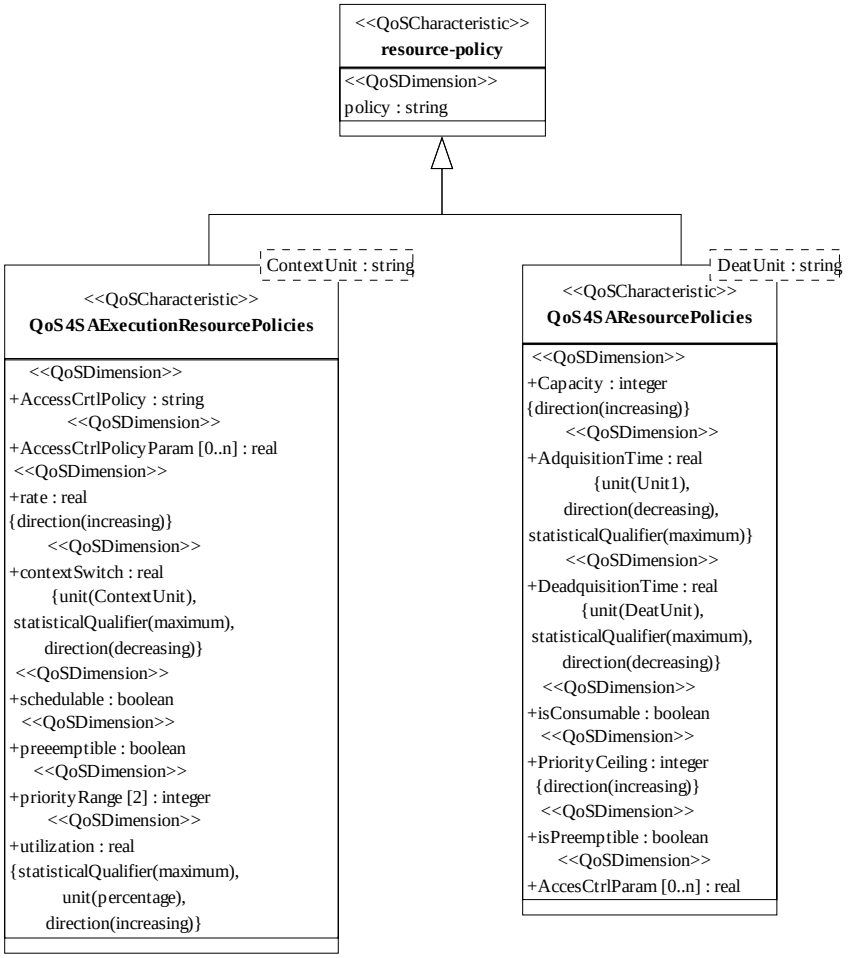

Figure B.4 - Policy Characteristics for Scheduling Analysis

B.2 Description of Telemetry Example with QoS Contracts

In this section we are going to use the QoS Profile presented in Chapter 9 and the QoS characteristics included in Section 10.3 to redesign the example included in Section 7.2.3 in [20]. This new design has removed all the UML extensions that were included in the original example, and we have used the QoS profile and the QoS4SA characteristics to represent similar concepts.

We start creating the *Quality Model*, which we are going to use in this example to represent any quality value. Figure B.5 includes QoS characteristics specific for this system, which resolves all parameters of characteristics templates. In this case we resolve all temporal units with the unit ms (all temporal expressions are milliseconds). The characteristics are:

- `TelemetryQoS4SADemand`,
- `TelemetryQoS4SAGlobalLatencies`,
- `TelemetryQoS4SAExecutionResourcePolicies`,
- `TelemetryQo4SAActivityResourceUtilization`, and

- `TelemetryQoS4SAResourcePolicies`.

They bind the characteristics included in Section B.1, “Scheduling Analysis Based on QoS Characteristics,” and resolve the parameters of characteristics.

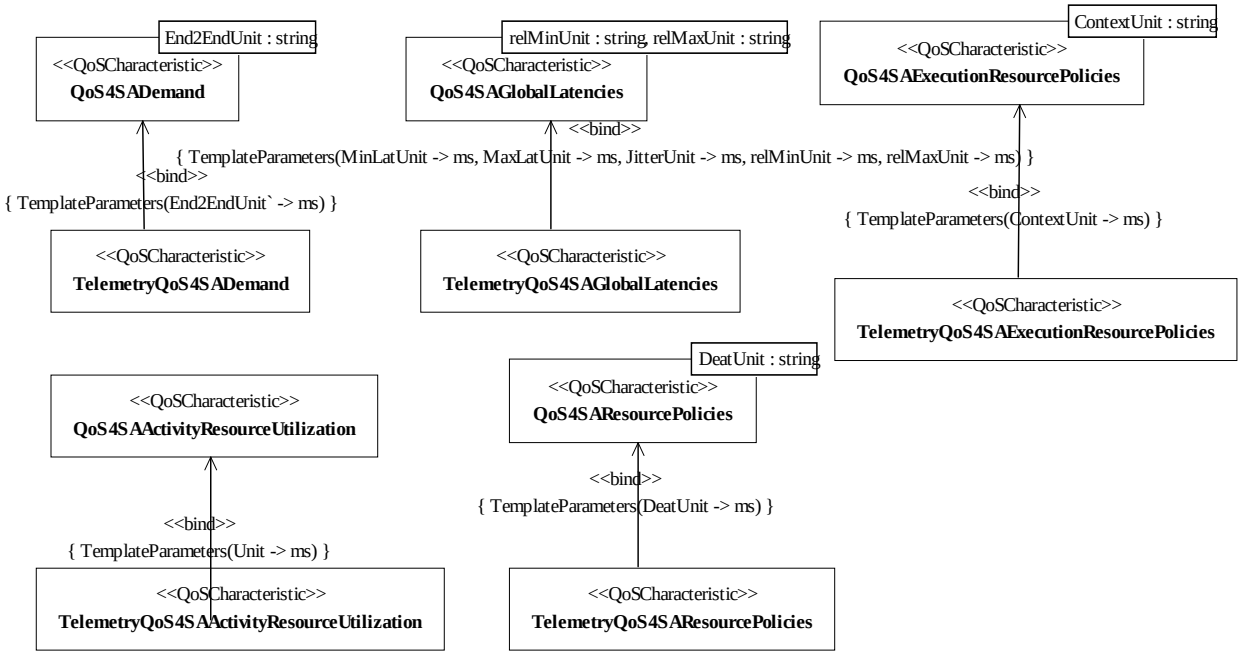

Figure B.5 - Telemetry System Quality Model

Figure B.6 shows the structural specification of a telemetry system example that includes the Figure 7-4 in [20]. This class diagram is a descriptor diagram, to enable schedulability analysis we must concrete the instances and physical elements that will support the system execution.

Figure B.7 is a UML 2.0 communication diagram associated to a package that includes the instances that represent the scheduling scenario. This diagram represents concepts equivalent to Figure 7-6 in [20], but it uses *QoS Values* and *Constraints* based on the *Quality Model* that includes the Figure B.5. The messages from `TGClock` are constrained with the period of demand and the latencies. `TelemetryGather` and `TelemetryProcessor` are annotated with Constraints and `TelemetryDisplay` is annotated with *Abstraction* dependencies that have associated *QoS Values* that describe the latencies and demand of this message.

We have included in Figure B.7 the resource of type Ix86Resource (it represents the CPU) that UML 2.0 communication diagrams cannot include, but we have included it to represent the dependency of some actions (A1.1 and A1.1.1) of this resource. The rest of actions depend on this resource too, but the dependency is not visible. The dependency is annotated as *GRMRequires* and *QoSContract*. The attribute *AllowedSpace* of stereotype *QoSContract* references the *QoSValues* that enumerate the quality that must provide the service of the resource.

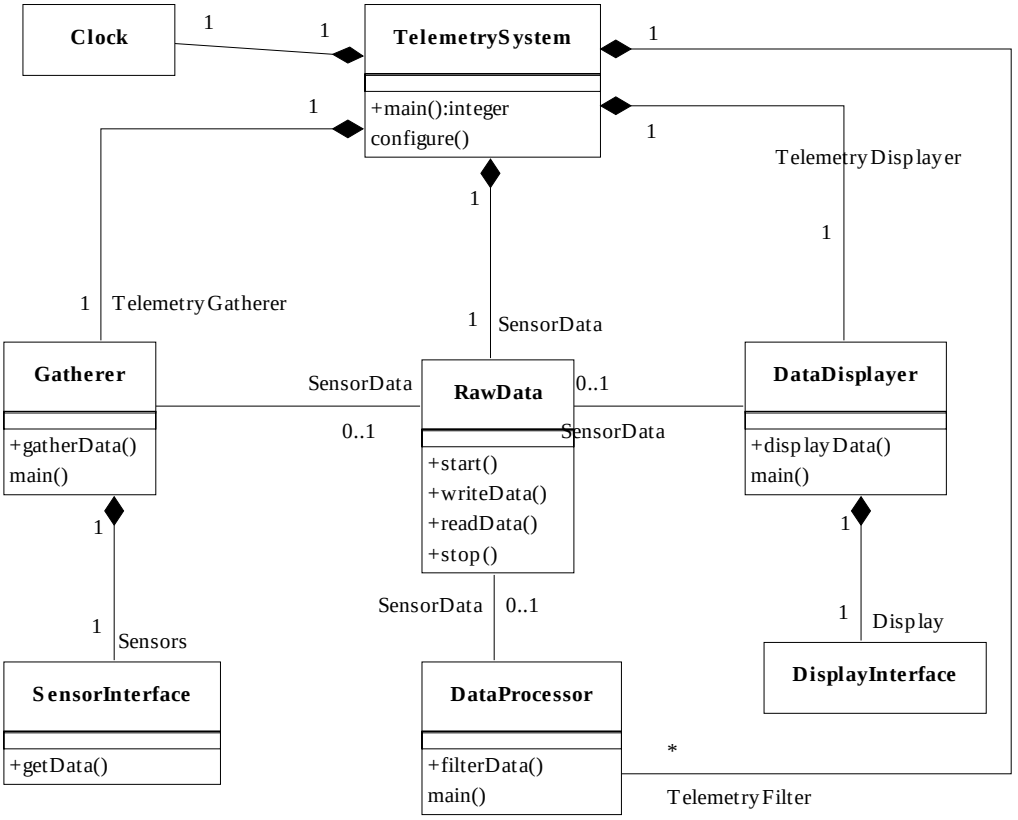

Figure B.6 - Logic Model of Telemetry System

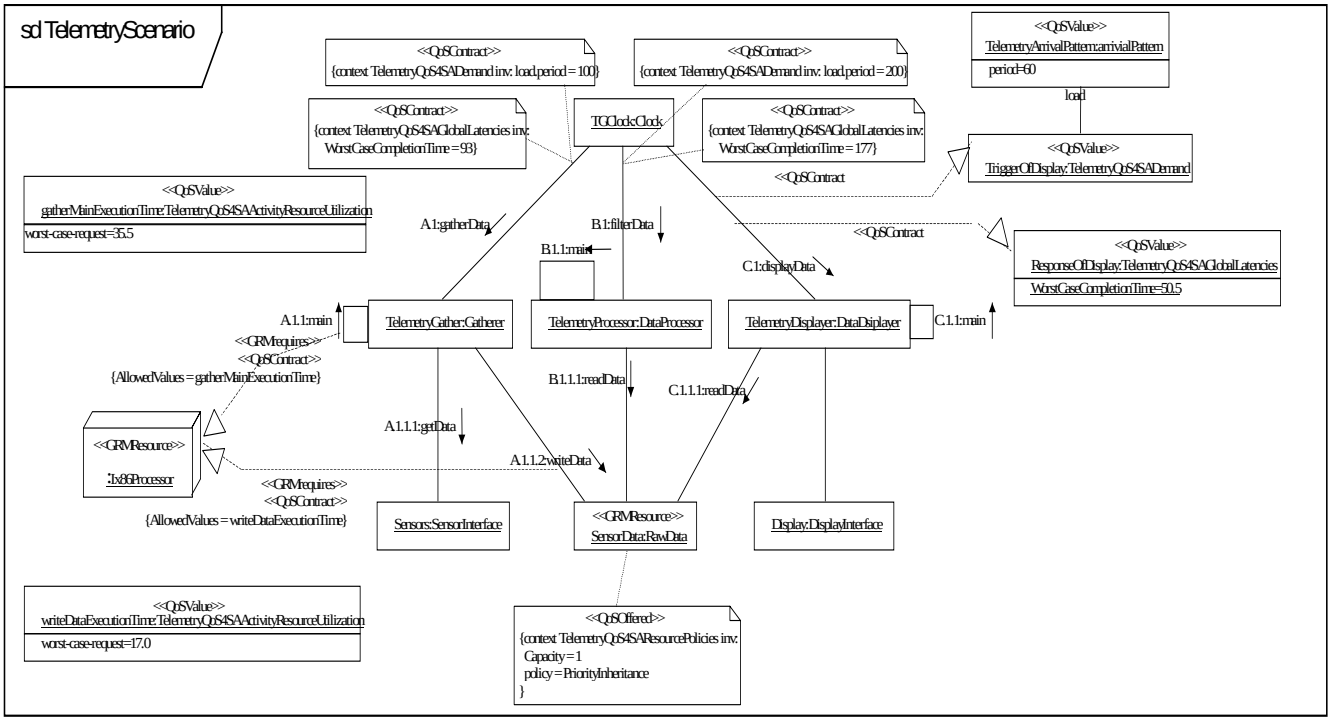

Figure B.7 - Expression of Schedulability Using Communication Diagram

QoS Offered constraints in Figure B.7 and Figure B.8 describe the QoS policies of the data and execution resources.

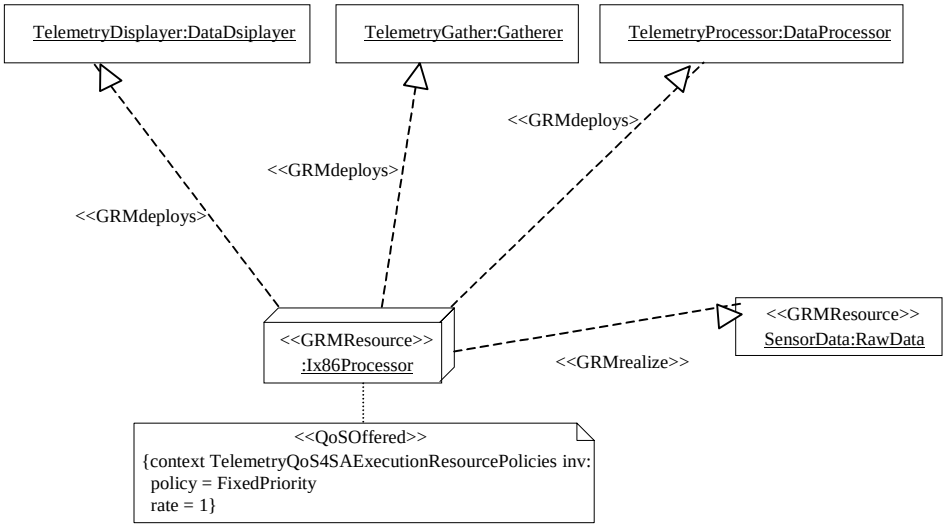

Figure B.8 - Execution Engine Policies

As in [20] the values of attributes in *QoS Values* and *QoS Constraint* can be resolved for the automatic generation of analysis models. To resolve these values the expressions of OCL expressions are limited and must be valuable statically.

Annex C: References

(informative)

[1] J. Aagedal Quality of Service in Development of Distributed Systems, PhD Thesis, Department of Informatics, University of Oslo (March 2001).

[2] J. Aagedal and E. Ecklund, “Modelling QoS: Toward a UML Profile”, Proc. <<UML-2002>> Conference, Springer Verlag 2002

[3] AS/NZS 4360:1999. Australian Standard: Risk Management. Standards Association of Australia, 1999.

[4] M. Barbacci, T. Longstaff, M. Klein and C. Weinstock., *Quality Attributes*, CMU/SEI Technical Report No. CMU/SEI-95-TR-021 ESC-TR-95-021, (December 1995).

[5] C. Becker and K. Geihs. MAQS - Management for Adaptative QoS-enabled Services. Proc. Workshop Middleware for Distributed Real-Time Systems and Services, IEEE Computer Society, (December 1997)

[6] M. Born, E. Holz and O. Kath, “A Method for the Design and Development of Distributed Applications using UML”, Proc. International Conference on “Technology of Object-Oriented Languages and Systems” (November 2000).

[7] M. de Miguel, J. Ruiz and M. García, “QoS-Aware Component Frameworks”, Proc. International Worshop on Quality of Service, (May 2002).

[8] DO, *Software Considerations in Airbone Systems and Equipment Certifications*, RTCA/DO-178B, RTAC, Inc, (1992).

[9] S. Frolund and J. Koistinen, “Quality of Service Specification in Distributed Object Systems”, Distributed Systems Engineering Journal, Vol. 5(4), December 1998.

[10] M. Gonzalez, J.Gutierrez, J. Palencia, J.Drake, “MAST: Modeling and Analysis Suite for Real-Time Applications”, http://mast.unican.es/

[11] L. Halton, “Exploring the Role of Diagnosis in Software Failure”, IEEE Software, Vol. 18(4) July 2001, IEEE Computer Society (2001).

[12] International Organization for Standardization, CD15935 Information Technology: Open Distributed Processing - Reference Model - Quality of Service, ISO document ISO/IEC JTC1/SC7 N1996 (October 1998).

[13] International Organization for Standardization, *Quality of Service: Framework*, ISO document ISO/IEC JTC1/SC 6 ISO/IEC 13236:1998 (December 1998).

[14] International Organization for Standardization, *Software engineering -- Product quality -- Part 2: External metrics I*SO/IEC TR 9126-2:2003 (July 2003).

[15] J. Koistinen, “Dimensions for Reliability Contracts in Distributed Object Systems”, Hewlett Packard Technical port, HPL-97-119 (October 1997).

[16] J. Laprie, “Dependable Computing and Fault-Tolerant Systems”, Depndability: Basic Concepts and Terminology in English, French, German, Italian and Japanese. Vol 5. Springer-Verlag (1992).

[17] N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley (1995).

[18] M. Lyu, editor, Software Fault Tolerance, John Wiley & Sons, (1995).

[19] Object Management Group, *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms RFP*, OMG document number ad/02-01-07 (January 2002).

[20] Object Management Group, *UML Profile for S*cheduling, Performance, and Time, Adopted Specification, OMG document number formal/2005-01-02 (January 2005)..

[21] C. Otero and I. Nitescu, "Quality of Service Resource Management for Consumer Terminals: Demonstrating the Concepts", 14th Euromicro Conference on Real-Time Systems, Work-In-Progress Session (June 2002).

[22] R. Pressman. *Software Engineering: A Practitioner's Approach* Mc Graw-Hill, Inc, (1997).

[23] M. Shankar, M. de Miguel, and J. Liu, "An End-to-End QoS Management Architecture", Proc. Real-Time Application Symposium, IEEE Computer Society (1999).

[24] W. Torres-Pomales, *Software Fault Tolerance: A Tutorial*, NASA/TM-2000-210616, (October 2000).

[25] N. Venkatasubramanian and K. Nahrstedt, "An Integrated Metric for Video QoS", Proc. ACM Multimedia 97, (November 1997).

[26] N. Wang, D. Levine and D. Schmidt, "Optimizing the CORBA Component Model for High-performance and Real-Time Applications", Proc. Work in Progress Workshop of Middleware 2000 Conference, IFAC/ACM (April 2000).

[27] J. Zinky, D. Bakken and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects", Theory and Practice of Object Systems, Vol. 3(1) 1997.

INDEX

S

T

U