Quality of Service Support in Development of Distributed Systems

by

Jan Øyvind Aagedal

THESIS

submitted to Department of Informatics,
Faculty of Mathematics and Natural Sciences, University of Oslo
in partial satisfaction of the requirements for the degree of
DOCTOR SCIENTIARUM

March 9, 2001

Table of Contents

Table of Figures

Abstract

This thesis investigates methodological support for Quality of Service (QoS) during software development. To this end, we define a general modelling language for QoS and show how it can be used in different types of software development activities.

QoS is a term that covers system performance, rather than system operation (i.e., functionality). For many categories of QoS, most parts of a system are involved in or influenced by the activities needed for the system to provide and maintain predictable QoS. The pervasive nature of QoS-support suggests that decisions on QoS-support should be an integral part of software development and not just a matter of configuration at deployment time.

Software development is in many cases initiated as a result of an enterprise modelling activity in which the need for computerised support is discovered. During enterprise modelling, the area of interest is identified and described, often resulting in requirements for a software system. In this thesis we show how enterprise modelling can be performed using the Unified Modeling Language (UML) and how the results of enterprise modelling can be used as input into the software development process. In particular, in order to be able to capture QoS requirements as part of enterprise modelling, we position QoS in the enterprise language of the Reference Model for Open Distributed Systems (RM-ODP).

In this thesis we also show how specification of QoS can be integrated in system specification. The major part of this work is the definition of CQML, the Component Quality Modelling Language. CQML is a lexical language designed for specification of QoS. Using CQML, the QoS a component provides can be specified independently of how the support is to be implemented. Furthermore, the QoS a component provides can be specified without affecting the specification of its functional properties. This means that, if needed, specifications of existing systems with already defined functional properties can be extended to be QoS-aware, without changing any existing part of the specification.

We show how CQML can be used in different parts of software development. In particular, we define a QoS profile for UML based on CQML that we position in a software development process by using it in a case study for a Lecture-on-Demand system. To provide computational support for QoS-aware systems that have specifications in CQML, a QoS framework architecture is defined and links to implementations of parts of it are provided.

Preface

"We judge ourselves by what we feel capable of doing,
while others judge us by what we have already done."
– Henry Wadsworth Longfellow, 1807-1882, "Kavanagh", bk. 1, ch. 1, 1849

This is a doctoral thesis submitted to Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo in partial satisfaction of the requirements for the degree of Doctor Scientiarum (dr. scient.). The work reported in this thesis has been carried out at SINTEF Telecom and Informatics, Department of Distributed Information Systems under the supervision of associate professor dr. Arne-Jørgen Berre, from September 1996 to March 2001. During this period, seven months (February to September 1998) were spent as a guest scientist at the Distributed Systems Technology Centre (DSTC) at the University of Queensland, Brisbane, Australia.

The work has been carried out in the context of the OMODIS project, a five-year project funded by the Research Council of Norway (NFR). The OMODIS project is a collaboration between SINTEF and Center for Technology at Kjeller (UniK).

While a PhD-study unquestionably gives in-depth and detailed knowledge about the subject matter investigated, I have learned other lessons I believe may prove at least as important in my future career. Firstly, I believe I have learned *to do research*. The growing acknowledgement and understanding of the importance of a scientific approach and an explicit research method is a valuable insight I think will become useful in future work; an analytical approach can be used in many areas of life. Secondly, I believe I have learned *to be precise*. Precision is important when reporting results to avoid any ambiguities and misunderstandings. Whether I have sufficiently learned these two important skills and used them in this thesis are up to the reader to judge. For most readers, however, my mastering of these skills is hopefully subsidiary to the contribution in the area of investigation, i.e., QoS-support in development of distributed systems.

Acknowledgements

I want to express my sincere thanks to my advisor, dr. Arne-Jørgen Berre, whose visions and technological overview never fail to amaze. I also want to thank the other senior members of the OMODIS project, prof. dr. Vera Goebel (project leader) and prof. dr. Thomas Plagemann, who have contributed both to the project and with useful comments on my work. In particular, I am profoundly grateful to the senior guest scientists in the project, dr. Earl F. Ecklund, Jr. and dr. Denise J. Ecklund, who both have spent much time and effort discussing and giving feedback on many of my thoughts and ideas during the final year of this work. They have been a very useful source of both encouragement and constructive criticism, and have been crucial to the result. I am also grateful to fellow project PhD-student, Ketil Lund, who has been a discussion partner for parts of the work and has contributed to an early version of the overall design of the QoS framework presented herein.

Many of my former and current colleagues at SINTEF have given me good support during the work. These include, but are not limited to, Bjørn Skjellaug, prof. dr. Ketil Stølen, dr. Steinar Carlsen, dr. Christen Krogh, Jon Oldevik, Arnor Solberg, and Anton Landmark. In addition, former and current MSc students under my supervision have performed useful work in relation to this thesis. In particular, Tom Christian Paulsen, who implemented the CQML parser and the QoS repository, and Lars Sveen Lundby, who is currently working with the QoS framework, should be mentioned.

My visit at DSTC was very rewarding, and I want to express my gratitude to dr. Kerry Raymond, Andrew Wood, dr. Andry Rakotonirainy, and in particular dr. Zoran Milosevic together with whom many new insights were gained.

Finally, I want to thank my family for good support during these years. In particular, I want to express my deepest thanks to my wife, Unni, for her patience and encouragement, and for always giving me the support I needed.

Part I. Overview

Chapter 1 Introduction

1.1 Motivation and Background

Traditional system development methodologies are mostly general purpose. They contain what is believed to be a suitable set of modelling elements that may be tuned to the specific problem at hand. The degree of flexibility in choosing a relevant set of modelling elements varies between the methodologies, from the strict approach where everything is predefined and under rigid control, to a flexible approach with few guidelines and no uniform terminology. In [Harmsen et al., 1994], the degrees of flexibility are illustrated as shown in Figure 1.

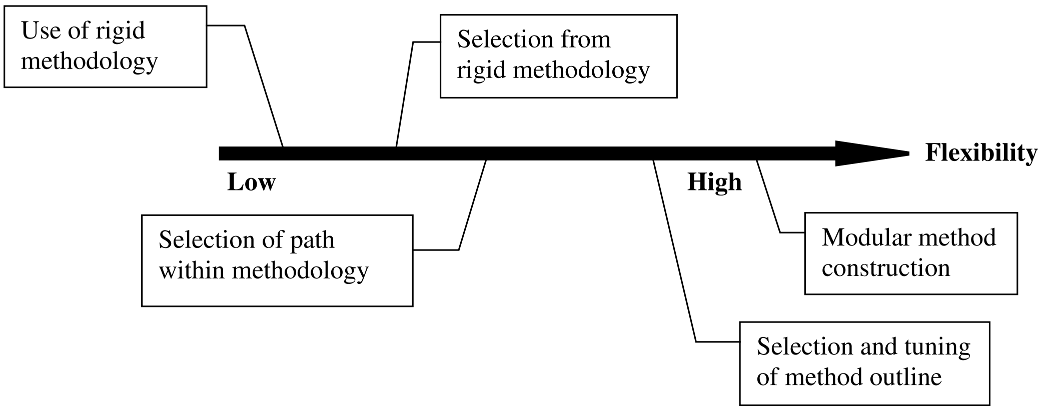

Use of rigid methodology
Selection from rigid methodology
Flexibility
Low
High
Selection of path within methodology
Modular method construction
Selection and tuning of method outline

Figure 1: Degrees of flexibility [Harmsen et al., 1994]

Methodologies can be characterised on a scale from no flexibility with a high level of control to a high level of flexibility without control. On the one extreme, an approach characterised by flexibility without control can hardly be called a methodology since any systematic and co-ordinated working methods are absent. On the other extreme, approaches characterised by rigid control without any flexibility usually do not fit the circumstances at hand, and pose restrictions that unnecessary restrict system developers. The palatable solution is to provide flexibility that can be used to tune the methodology to fit the particular problem, but still provide guidelines and process support the system developers can utilise in their work.

A prerequisite to create such a system development methodology, is that the repertory of modelling elements, from which the specific methodology is created, contains what is needed. This means it must be possible to specify whatever construction is needed in that particular domain at the appropriate level of abstraction. In theory, a Turing machine can emulate any computation, but the level of abstraction is not appropriate for practical system development. A system development methodology should abstract the underlying computational model so that system developers can focus on the problem at hand, not on details of its solution. In the domain of distributed systems, which is the focus of this work, many inherent characteristics such as concurrency, distribution, real-time, etc., influence what must be appropriately supported in the methodology.

Traditionally, the goal of system development methodologies is to support analysis of particular, isolated problems and the design of their solutions. The results of using such methodologies are often large, monolithic systems. The organisational trends on downsizing and virtual enterprises and the advances in networks give the need for development of flexible, often distributed systems, supporting these new structures. As technology for distributed objects such as CORBA [OMG, 1996] from Object Management Group (OMG), Microsoft's COM+ [Rogerson, 1997] and others have received wide market acceptance, methodologies for distributed objects are becoming more important. Based on these trends, the goal of many system development projects is to develop applications utilising flexible architectures based on components using distributed object technology.

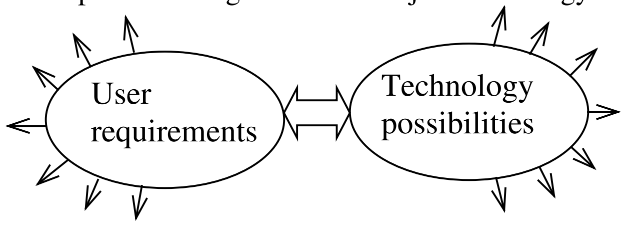


Figure 2: Mutual influences in technology development

Figure 2 depicts the mutual influences between user requirements and new possibilities in the technology, and their ever-increasing nature. As the technology advances, it paves the way for new applications utilising the new features. Likewise, as the users see new possibilities, they require new applications that need even more advanced support from the technology. With the constant advances in the hardware area, multimedia features, for instance, are incorporated into many applications. Multimedia features combined with networks make distributed multimedia applications more and more mainstream technology. The development of such applications needs methodological support. A methodology suited to develop this kind of applications needs support for both distribution and multimedia, features traditionally not focused on in system development methodologies.

Improvements in network and computing technologies are continually extending the potential of distributed systems. However, developing distributed systems remains expensive and error-prone. This situation has led to a "distributed software crisis" where the basic network and computing technology continually improves while distributed software using this improved technology gets bigger, slower and more expensive, both to develop and maintain [Schmidt, 1997]. Given the advances in basic technology, users increase their expectations of what can be achieved. In addition, the technology advancements facilitate new application types like multimedia, simulation, command and control and others, so both the end user pull and the technology push contribute to the emergence of novel distributed applications. In addition to correct functionality, users of these kinds of systems often require guarantees of quality of service (QoS). This means an appropriate level of quality of service must be specified and guaranteed.

Quality of Service (QoS) is a general term that covers system performance, as opposed to system operation. Support to manage QoS is emerging in infrastructure components such as networks and operating systems, but there has been little attention to how this support should be reflected in software development methodologies. Apart from real-time methodologies that address issues of timeliness, most system development methodologies focus on the specification of system operation, ignoring system performance. However, QoS is crucial in multimedia applications, and methodological support for QoS is needed when developing applications in this domain.

1.1.1 OMODIS

The work reported herein was done in the context of OMODIS. OMODIS (Object-Oriented Modeling and Database-Support for Distributed Multimedia Systems) is a five-year (1996-2001) collaborative project between Center for Technology at Kjeller (UniK) and SINTEF. OMODIS is part of the DITS program (Distributed Information Systems), a strategic basic research program supported by the Research Council of Norway. The focus of OMODIS is on modelling and database support for distributed multimedia systems.

1.2 Research Method and Research Topic

Research is an activity where one starts with an open issue and tries to produce some new insight on this following a scientific approach. What scientific approach to follow strongly depends on which issues to investigate and in which field of science the research is carried out. In the area of computer science, and in particular the field of software methodologies and software engineering, there is no well-established research method, both due to the diversity of research issues and to the youth of the field. Some research topics in this field (e.g., where performance is an issue) are well suited to a traditional deductive approach based on empiricism where hypotheses with clear falsification criteria can be identified. However, there exist areas where the traditional use of the deductive approach with empirical justifications is less obvious. Many parts of computer science are synthetic disciplines in the sense that they investigate phenomena which are not part of nature, but entirely made by humans. Many parts of computer science focus not on investigating these human-made phenomena as such, but rather on the processes used to create them. This is especially valid in the area of software methodologies where one defines concepts and the use of these to abstract phenomena sometimes only present in software. Research in this area often involves creation of new abstraction mechanisms that are meant to help software developers, both to reduce development efforts and to be able to understand and model complex problems. However, software modelling is an individual and context-dependant activity, concepts that help some may be of no use to others. Empirical studies of the use of these new concepts are therefore very difficult (let alone very costly) to perform; there are too many variables out of control to draw any conclusions from most of such studies. Hence, in these areas of computer science, the deductive method is often used without much empiricism.

In purely theoretical research, e.g., in parts of mathematics and logic, axioms are defined, theorems hypothesised and then proven, and the results interpreted. This analytical approach is of particular use when the work is done within a sound theoretical framework where a basic set of axioms and some deductive rules are defined. However, many areas of computer science do not exist within a formally defined theoretical framework. Nevertheless, the analytical approach to the deductive method is often applicable. In many of these cases, new concepts are introduced based on some underlying set of basic concepts, and the relationship between these new concepts and the existing ones are pointed out. The overall hypothesis is normally that these new concepts are useful in some context, i.e., they have more expressive power, represent some new abstractions, or represent new relationships between formerly unrelated concepts. This hypothesis can be validated using several approaches. Firstly, case studies can be used to justify the hypothesis by showing how complex problems that until then were too complex to solve, now can be solved. These are so-called proof-of-existence experiments where the new concepts make a solution possible. Secondly, by comparison with existing relevant sets of concepts, it can be more or less formally shown that the new concepts represent ideas until then hard to represent. Thirdly, case studies can also be used to show that manageable problems get a simpler solution. These are so-called proof-of-concept experiments where it is justified that the new concepts satisfy some objective, e.g., they can help to hide unnecessary details. Finally, based on the new set of concepts, concepts that usefully relate existing abstractions can be derived. Example of this last type of validation is to show that the modelling concepts introduced are useful when mapping a software design to existing infrastructures (hardware and software platforms). So-called proof-of-performance experiments where measurements are made are hardly used in the area of software methodologies due to the difficulties of performing empirical studies outlined above.

ACM's taxonomy of computer science presented in [Denning et al., 1989] divides research in computer science into three different paradigms and nine different subject areas. The work reported herein is in the subject area of Software Methodologies and Engineering and focuses on the theory and abstraction paradigm. Given this area of concern, the deductive method without traditional empirical studies is used.

The focus of this work is the development of QoS-aware distributed systems. To this end, we carried out the following steps:

1. *State-of-the-art investigations.* As new ideas and solutions are continuously being put forward, the state-of-the-art changes accordingly. To investigate the state-of-the-art is therefore an ongoing activity. However, an initial investigation can reveal key players in the field and important conferences and journals to monitor carefully, so by narrowing the scope to these sources of information, the task of keeping up-to-date can be manageable. By reducing the scope of what publications to read and which places to search for new ideas, the obvious risk of missing good and novel pieces of work is present. It is therefore important to not completely forget other sources of information. After the initial identification of key sources of relevant information, we tried to reduce this problem by doing brief periodic searches using search facilities on the Internet or library search facilities such as IEEE/IEE Electronic Library [IEEE/IEE, 1999], INSPEC [IEE, 1999] or ACM Digital Library [ACM, 1999]. However, we cannot feel certain that all relevant work was considered when doing the research presented herein, but we tried to address this problem as well as was seen fit. The trade-off between continuous search for new and relevant ideas and advancing your own must be balanced, and we hope we managed this satisfactory. State-of-the-art is documented in Part II that constitute the main part of this work.

2. *Problem identification.* Based on initial state-of-the-art investigations and on communication with fellow researchers, the problems addressed in this thesis were identified. The nature of a problem to investigate in this kind of work is twofold: it must be a real problem to which a solution will advance the field, and it must be general so that a solution will help in the general case, not just for some specific applications. To narrow down a wide field of interest to specific problems to investigate is a real challenge. From our general interest in software development and distributed systems, we chose to focus on support for QoS. Most software development methodologies focus only on functional aspects of the system, leaving non-functional aspects to be incorporated implicitly during design or as a matter of configuration during deployment. In many applications such as multimedia and command and control systems, the level of QoS is very important for user acceptance. Therefore, we wanted to include QoS-awareness into a methodology for distributed systems so that conscious decisions can be made about QoS during software development.

   We decomposed this overall problem into a number of subproblems that were addressed by seeking answers to the following questions:

   1. How can QoS be positioned in general enterprise modelling?
   2. Based on the assumption that a well-defined enterprise language in RM-ODP is useful for positioning QoS in enterprise models, what is the mapping between the enterprise language in RM-ODP and the UML?
   3. How can QoS be positioned in a general software development methodology for distributed and component-based systems?
   4. How can QoS be expressed using UML at different levels of abstraction?
   5. How can QoS be expressed in a lexical, declarative language and positioned together with interface and component definition languages?
   6. What support is needed from the computational platform to enable QoS-awareness in systems running on it?
   7. How can the lexical language be mapped onto a computational platform that supports QoS?

   The first two questions address problems to which solutions enable us to capture system requirements and to have a representation of the system in its enterprise context.

Enterprise modelling is an area of wide and increasing interest, as both the Enterprise Distributed Object Computing (EDOC) conference series and the focus in OMG on enterprise solutions show. With a background knowledge of role-modelling as presented in OOram [Reenskaug et al., 1996], we saw the need for a consistent and sound set of concepts that positioned the concepts of roles and role models in enterprise modelling. Also, we identified the need to position QoS within the overall enterprise modelling approach.

The next two questions address the problem of including QoS in a software development methodology. While enterprise modelling focuses on business concepts and may lead to an identification of system requirements, we also need support when developing specific information systems within such an enterprise to meet the identified requirements. At SINTEF, we have over a number of years developed and refined a methodology for developing component-based systems called COMET (Component Development Methodology). This methodology focuses mainly on functional aspects of the system, leaving QoS to be considered only implicitly. With the use of COMET in areas such as command and control systems, we saw the need to include a conscious modelling approach to QoS. One part of this problem is to be able to express QoS-aspects in the de facto modelling language UML used in COMET, another part is to identify in which models and when QoS is to be considered. We address both problems.

In addition to modelling support for QoS in UML, we chose to focus on a lexical modelling language for QoS. For general system development, interface definition languages of various kinds are extensively used to model functional properties of a system, but there is no appurtenant language to model QoS aspects of a system. In the early phases of our work, it was made clear that such a language was needed and we identified this as an area of research [IWQoS '97 panel, 1997, ICODP '97 position paper session, 1997]. The fifth question addresses this problem.

Related to QoS, mechanisms in the computational infrastructure that support QoS management are outside the scope of this work. This means that QoS support in infrastructures (e.g., middleware and network technologies) and appurtenant mechanisms for QoS are not addressed herein. However, we need to define a computational model in which QoS has a part so that the concepts used in the methodology can be mapped onto a computational platform. The last two questions address this problem.

Our hypothesis is that support in the methodology for modelling QoS will help the development of advanced distributed systems such as multimedia systems and others. Such support can help developers in two ways: i) by introducing new abstractions so that developers can use high-level constructs to model QoS aspects during the different stages of the software development process, and ii) by having direct mappings from the modelling constructs to QoS-enabled infrastructures. In this work we investigate both areas. As a basis for our work, we used the following premises:

- QoS is essential for application developers in some domains, specifically in the distributed multimedia domain.
- ISO Reference Model for Open Distributed Processing (RM-ODP) [ISO/IEC JTC1/SC21, 1994, ISO/IEC JTC1/SC21, 1995a] represents a consistent framework for description of open distributed systems.
- UML [UML RTF, 1999, ISO/IEC JTC1/SC7, 2000b] represents state-of-the-practice of modelling languages.
- QoS is supported in infrastructures and is specified using low-level constructs, but constructs on the appropriate level of abstraction for software designers are absent.

3. *Solution proposal.* Based on the problems identified and state-of-the-art investigations, we created a set of new constructs that address the issues raised. The details of the proposed solutions constitute the main body of this thesis and are reported in Part II and Part III, and also in the papers included in appendix I.

4. *Proof-of-concept.* To justify that our proposals represent possible solutions to the problems we have identified we performed several activities.

   First we argued what problems existing approaches were experiencing with respect to QoS, and how our proposals can contribute to solutions of these. As part of this argumentation, we highlighted the points we tried to make by small examples. This informal argumentation is given along with the proposed solutions.

   Second, to validate the proposed modelling techniques, we used them in a case study of modelling a Lecture-on-Demand (LoD) system. An LoD-system is an advanced multimedia system where QoS is of great importance. Hence, modelling of such a system represents a typical example of the topic of concern. We use this case study to investigate whether the proposed techniques are appropriate. Specifically, we show how QoS can be modelled at the end-user level so that LoD system requirements extend to include QoS aspects. We also show how to transform QoS aspects in the LoD-system from the user level down to the design level during the software development process. It is important that the model refinements originally in COMET extend to include QoS aspects, and we show how to do this. As part of this, we show that the modelling constructs introduced in this work are suitable to represent the QoS aspects on all levels of the LoD case. We also identify constructs in a QoS framework that can support the modelling constructs used in the case study.

   Thirdly, we argue that the proposed solutions can be used in other areas, namely those of process assessment and requirements engineering. Without performing case studies, we relate the issues relevant in these areas to the proposed solutions, and argue informally how these can alleviate some of the problems.

   Finally, we have initiated work on implementing mappings from the lexical modelling language to a supporting computational infrastructure, and on implementing the computational infrastructure itself. The compiler for the lexical language and a run-time representation of the language are completed, while work on implementing other parts of the infrastructure is ongoing.

1.3 Thesis Overview

This thesis is divided into four parts. The first part, that includes this introduction, gives an overview of the thesis and outlines problems and requirements. It also provides an overview of the concept of Quality of Service in Chapter 2. The second part focuses on modelling of QoS and Chapter 3 provides an overview of terminology before the main chapter of this thesis, Chapter 4, presents the Component Quality Modelling Language (CQML), a lexical modelling language for QoS. CQML can be used in addition to an interface or component definition language (IDL/CIDL) to be able to specify QoS-aware, distributed components. Chapter 5 follows up on this by discussing how CQML can be used in different parts of software development. As part of this, a UML profile for QoS is defined based on the concepts of CQML, and results of using CQML in a case study is reported. The third part outlines how computational support for the concepts in the previous part can be provided. In this part, Chapter 6 presents a QoS framework architecture in which CQML specifications are used as input to perform QoS management. The fourth and final part contains a discussion of novelties and issues raised, and it offers some critical comments on the work and points to possible future work.

In addition, appendix I reports four published papers on different aspects of modelling. Two of the papers are on enterprise modelling using UML and the positioning of QoS in enterprise modelling, respectively, whereas the third paper reports an approach to integrate QoS in UML-based modelling. The fourth paper addresses QoS specification and proposes a lexical language for this. Appendix I is the bibliography, while appendix III contains the CQML grammar and appendix I the IDL for the run-time representation of CQML.

The structure of the thesis is shown in Figure 3, along with possible reading tracks. As an alternative to a linear reading approach, the well-informed reader may skip Chapter 2 and/or Chapter 3 before reading Chapter 4. Chapter 4, however, is the main chapter of this thesis and cannot be skipped. After Chapter 4, Chapter 5 and/or Chapter 6 may be read before Part IV summarises and concludes. The papers in appendix I are stand-alone papers and can be read independently.

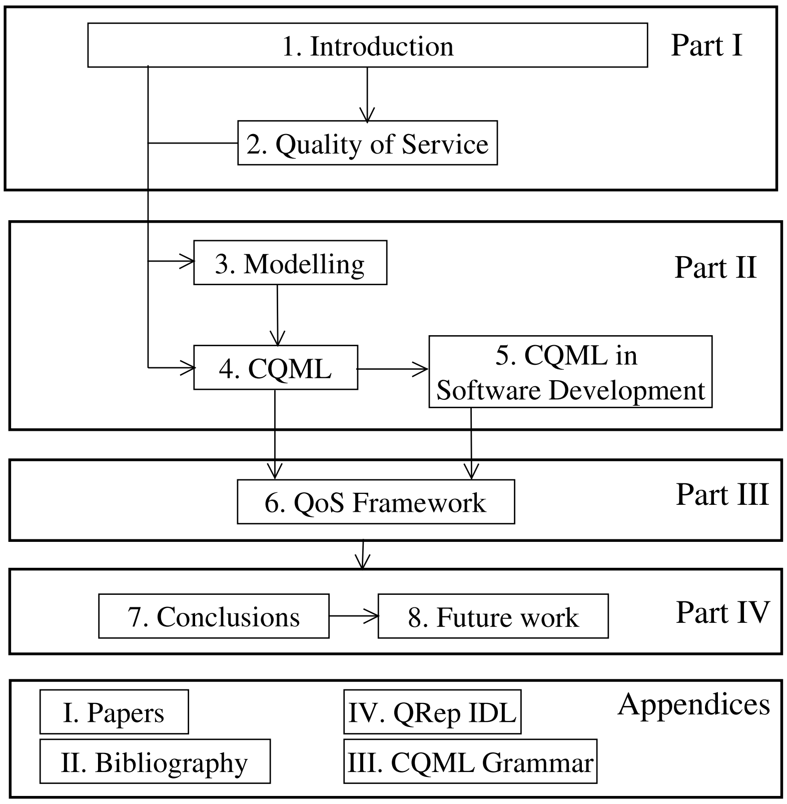


Figure 3: Thesis overview

Chapter 2 Quality of Service

This chapter discusses the term Quality of Service by first providing a discussion of quantification in general and then presenting a number of definitions and some terminology used in the area of Quality of Service. Different abstraction levels for QoS are presented, and modelling and architectural support are discussed.

2.1 Overview

Systems are primarily made to meet some functional specification, that is, to exhibit a particular input and output behaviour. However, systems have additional properties that characterise the systems' ability to exist in an environment and adapt as its environment varies. The functional properties of a system relate to the specification of what the system should do. In addition, the extra-functional properties, also called qualities, address how well this functionality is (or should be) performed if it is realised. In other words, if an observable effect of the system doing something can be quantified (implying that there is more than 'done'/'not-done' effect of the behaviour), one can describe the quality of that behaviour.

2.2 Quantification

The quantification of quality can be against some commonly agreed reference unit, like delay in seconds, or by ordinal rankings, like a scale from 1 to 5. The former is often referred to as an objective measure, while the latter is a subjective measure.

Quantification and classification are not simple. Agreeing on reference units (scales) and classification of observations onto such scales are important issues discussed in philosophy. Philosophers devoted to the issue of "vagueness" have been preoccupied by the sorites paradox for ages. The sorites paradox can be illustrated by the classical example of a pile of wheat and the predicate "is a heap". Clearly, one grain of wheat is not a heap. Also, adding a single grain of wheat to something that is not a heap does not make it a heap, i.e., two quantities of wheat only differed by a single grain are indistinguishable with respect to their "heapness". However, one million grains of wheat clearly makes a heap. Since one grain is not a heap but a million grains is, the second assumption (induction step) must be investigated and this is where the vagueness resides. Vagueness can be compared with ambiguity and generality. Relativisation disambiguates (the inaccurate predicate "is tall" can be relativised to "is tall for a 20th century male Norwegian"), but it does not eliminate borderline cases (if a tall, male, living Norwegian was 1 cm shorter, would he not be tall?). Generalisation helps to leave out details, hence more objects can be characterised, but this does not eliminate vagueness. To explain vagueness, one theory is "vagueness-as-ignorance", supported by epistemologists, where they state that precise dividing lines exist and vagueness is a result of not having the knowledge to establish these sharp lines. Others try to address the sorites paradox by acknowledging the existence of a grey area in between the two extremes. These approaches include deviant logics such as fuzzy logic, intuitionism, paraconsistent logic, etc. We take a pragmatic approach to these issues. Following [Vieru, 1998], we assume a stability threshold can be identified above/below which a characterisation is commonly agreed. Note that a stability threshold does not represent an inherent, precise and objective dividing line as assumed by epistemologists. With our assumption, it makes sense to characterise measures, even if inherently vague.

Vagueness is an issue both for subjective and objective measures. In particular, when observations measured on a continuous scale such as time are mapped onto a discrete (albeit objective) scale, vagueness appears. If, for instance, we want to map elapsed time onto the number of whole seconds from initiation, we can use common rules to round off to decide which of two adjacent seconds to choose. With respect to subjective measures, relativism is also an issue to consider. What one may characterise as "good" may be characterised as "average" from another. Actually, what one may characterise as "good" at one point in time may be characterised as "average" at another point in time by the same observer. In this case, the stability threshold is what everyone (in the group of evaluators) would consider "good" for all practical purposes all the time. Note the difference between "all the time" and "for all practical purposes all the time". The first excludes any theoretical possibility for anyone to characterise the subject matter as "average" at any point in time, while the latter is the pragmatic version, acknowledging that this would never happen in real life, even though a theoretical possibility exists.

2.3 Definitions and Terminology

In [ISO, 1986], quality is defined as "*The totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs*". In this definition, "product or a service" may be the result of activities or processes, or an activity or process itself. In [Miriam-Webster, 2000], service is defined as "*the work performed by one that serves*". Combing these two definitions, quality of service relates to the degree of satisfaction given by work performed by a server. Numerous specific definitions of the term quality of service exist. The general definition from ITU [ITU-T, 1994] states that QoS is "*the collective effect of service performances which determine the degree of satisfaction of a user of the service*". A network-oriented view is reflected in the definition in [Fluckiger, 1995] where it says "*Quality of Service is a concept based on the statement that not all applications need the same performance from the network over which they run. Thus, applications may indicate their specific requirements to the network, before they actually start transmitting data.*" An even more network-centric view is reflected in [Hindin, 1998] that states "*QoS is a way to allocate resources in switches and routers so data gets to its destination quickly, consistently and reliably.*" ISO's definition in [ISO/IEC JTC1/SC21, 1995b], "*a set of qualities related to the collective behaviour of one or more objects*", is related to the Reference Model for Open Distributed Processing (RM-ODP) which describes a distributed system as a set of objects. [Vogel et al., 1995] presents a definition that relates to distributed multimedia systems: "*quality of service represents the set of those quantitative and qualitative characteristics of a distributed multimedia system that are necessary to achieve the required functionality of an application*". Despite the different definitions, quality of service is a general term that covers system performance, rather than system operation (i.e., functionality).

Our primary concern is quality of service, not quality in general. However, in order to ensure that a service has a certain quality, we need to consider the quality of the product delivering the service (i.e., the server). Actually, it is in most cases artificial to distinguish between quality of service and quality of the product delivering the service. For instance, a quality characteristic such as reliability is a characteristic of the server, but it may also be viewed as a quality characteristic of the service. Hence, product quality is relevant even if our concern is quality of service. Process quality, on the other hand, is in most cases not relevant when considering quality of service. Much work has been done in the field of software process improvement to enable assessment of software development processes (e.g., Capability Maturity Model (CMM) [Software Engineering Institute, 2000, Paulk et al., 1993] and the ISO/IEC 15504 [Kitson, 1997, ISO/IEC JTC1/SC7, 1998]). The underlying assumption is that the quality of a software product only gets as high as the quality of the process used to develop it. However, even if software process quality is important to ensure software product quality, it is outside the scope of this work. We focus only on quality of service and how to support modelling of this, not on assessment of development processes. Our objective is to assist software developers in obtaining the required level of QoS for the software product they develop, and we do this by establishing a set of modelling concepts they can use in their software development process. We do not specify how the use of these concepts influences the quality of the development process. Nevertheless, the concepts defined in CQML can be used to specify process quality; this is elaborated on in section 5.2 when the use of CQML in process assessment is discussed.

At the end of the day, it is the perceived quality for the end user that determines whether the QoS is sufficient for its purpose. The perceived quality can be defined as how well the intended message by the sender corresponds to the delivered message as perceived by the receiver. This is sometimes referred to as Quality of Message (QoM). Perceived quality depends on a number of factors, of which the technical solutions cater for only a few. QoS for software systems is our primary concern in this work, not perceived quality. However, perceived quality defines the end-user expectations that in turn defines QoS expectations towards the technical solutions, so perceived quality must also be considered when developing QoS-aware applications.

According to the ISO QoS Framework [ISO/IEC JTC1/SC21, 1995b], QoS characteristic is the most fundamental term to express QoS. A *QoS characteristic* represents some aspect of the QoS of a system, service or resource that can be identified and quantified. QoS characteristics are to be distinguished from QoS parameters. A *QoS parameter* is any value related to QoS that is conveyed between entities. QoS characteristics can be grouped into QoS categories. A *QoS category* represents a type of user or application requirement.

Expressing the total QoS in terms of a set of QoS characteristics facilitates reasoning about QoS. From the QoS in ODP working draft [ISO/IEC JTC1/SC21, 1998], there are five categories of *QoS statement* used when describing QoS (or its constituent QoS characteristics), and subsequently in building distributed systems:

i) *QoS requirements*, which state QoS constraints that users of the system or its components have on the system or its components.

ii) *QoS capabilities*, which state the actual abilities provided by the components or configuration of components in the system with regard to QoS.

iii) *QoS offers*, which describe advertised QoS.

iv) *QoS contracts*, which are the results of agreement processes.

v) *QoS observations*, which express the values of QoS characteristics as observed using measurement functions.

A more complete presentation of the terminology from the ISO QoS framework and QoS in ODP is given when defining CQML in Chapter 4.

2.4 QoS Abstraction Levels

An application specification sometimes includes promises of certain qualities of service. To keep these promises, the application requires some quality of service from its underlying infrastructure. However, the infrastructure is indeed a set of systems in their own rights. These systems have also specifications that include some promises of certain qualities of service, promises that need support from their underlying systems to be kept. This recursive structure implies that quality of service needs to be specified at different levels of abstraction, from the end user of the system down to the hardware components the system uses.

System specifications, that may include specifications of QoS, consist of models on different levels of abstraction. A QoS specification on a high level of abstraction can be refined to more concrete specifications, finally ending up with QoS specifications supported by the underlying infrastructure. Figure 4 illustrates different levels of abstraction for specification of QoS. The end user is mainly concerned with the perceived quality of service from the system as a whole, irrespective of whether the infrastructure or the application is the one providing it. An application may consist of subsystems and run on a target platform with an operating system running on top, all of which may provide services with some QoS that the application depends on in order to provide its QoS.

In many cases, a gap exists between the socket-level QoS offered by the communications infrastructure and the software development being done at the application level. QoS guarantees are often in network terms like bits per second, not in application terms such as invocations per second. However, work is being carried out to incorporate QoS in higher level frameworks such as RM-ODP, TINA, and CORBA [Sluman et al., 1997, Leydekkers and Gay, 1996, ISO/IEC JTC1/SC21, 1995b, ISO/IEC JTC1/SC21/WG7, 1997, TINA-C, 1994, Stefani, 1993, Coulson et al., 1994].

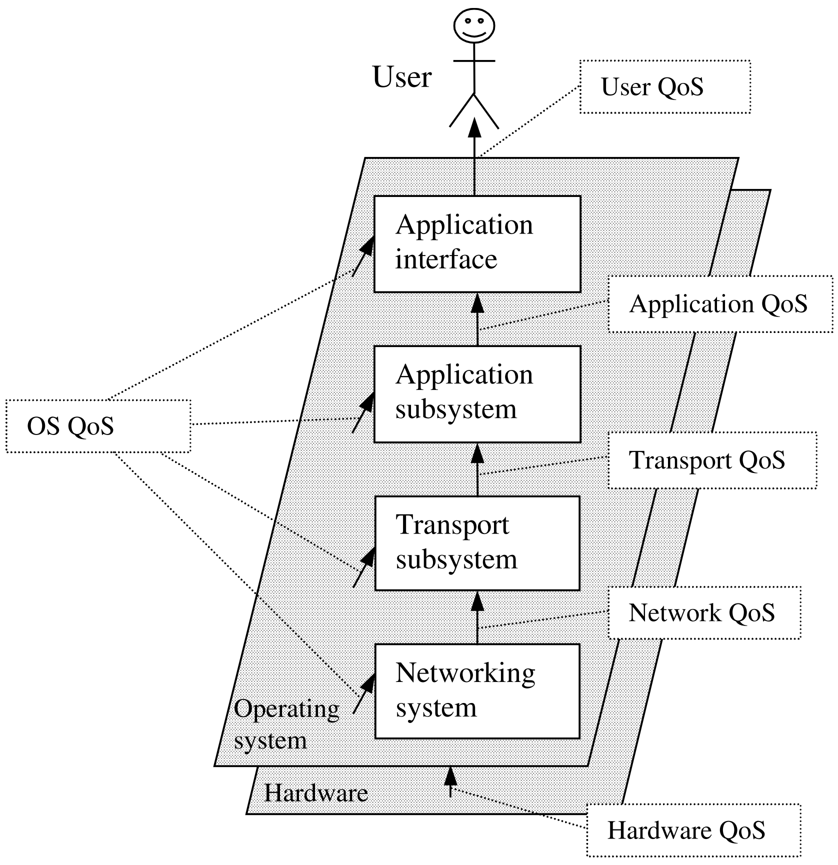


Figure 4: Different QoS abstractions

In [Leydekkers and Gay, 1996], different levels of QoS are organised according to the ODP viewpoints, and the result is shown in Table 1.

ODP Viewpoint	QoS level		Characteristics		Examples	
Enterprise	Subjective QoS		End-user oriented QoS requirements (subjective and not formal)		"The display should be clearly visible under all conditions"	
Information	Objective QoS		Precise statement of QoS requirements derived from the subjective QoS specification, often application independent.		25 fps(PAL), 30fps (NTSC) Telephone quality: 8 kHz	
Computational	Application QoS		Describe the QoS requirements for the applications often specified in terms of media qualities and media relations.		Updates of car position can be expressed in updates/s.	
Engineering	System QoS	Net-work QoS	Describe the QoS requirements from the operating system.	Describe the QoS requirements from the network.	Buffer size, Operations/s, Memory (Mb)	Throughput (Mbit/s), Jitter (Ms), Delay(Ms)
Technology	QoS properties		Describe the QoS characteristics of devices, operating system and network technologies, QoS properties.		Video device (PAL format) Audio device (μ-law) Cell loss rate for ATM	

Table 1: QoS levels related to ODP viewpoints [Leydekkers and Gay, 1996]

Table 1 shows that the ODP viewpoints address different aspects of a distributed system, thereby focusing on different QoS abstractions in each viewpoint. The enterprise viewpoint addresses *subjective* QoS dealing with QoS specifications being different for each end-user.

Such specifications are often more qualitative than quantitative, and they must be interpreted and translated into other QoS specifications in order to be meaningful for the system to provide them. In the information viewpoint, *objective* QoS attributes can be specified, often caused by QoS specifications in the enterprise viewpoint. The end-user specification "telephone quality sound" means the audio frequency must at least be 8kHz. In the computational viewpoint, computational objects are identified and *application* QoS characteristics like media qualities and relations between media are attached. In the engineering viewpoint, distribution issues are of concern, and the QoS specifications can be divided between *system* and *network* QoS describing QoS requirements like buffer size, memory and throughput, jitter, respectively. In the technology viewpoint, the QoS specifications for the hardware/software used to implement the system are specified (e.g., device QoS). It is important to note that the QoS specifications must be consistent between the different viewpoints, in addition to specifying new QoS attributes belonging to each viewpoint.

2.5 Infrastructure Support

QoS is generally pervasive across all system components, i.e., all system components contribute to the perceived quality of service. This means that the underlying infrastructure, comprised of hardware, network, operating system, middleware and other generic system components, needs to be QoS-aware in order to provide appropriate support for the applications.

Quality of service must be provided end-to-end from the information source to the information sink. The information source can be an input device or a storage device. The latter means that also the characteristics of the storage device, for example a database management systems (DBMS), must be included. Figure 5 indicates the levels where QoS decisions must be taken in a distributed system; DBMS issues may expand the source side if the information resides within a database[1].

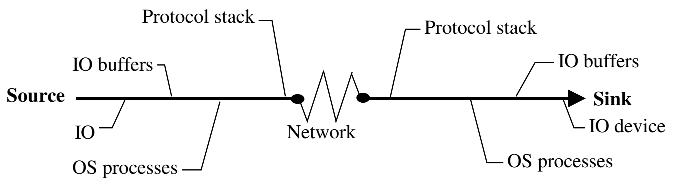


Figure 5: End-to-end view of QoS specification

The infrastructure needs to perform a number of activities in order to provide QoS-support for the applications. Such activities include QoS negotiation, QoS monitoring, QoS control, and others. QoS management is a general term for activities to support monitoring, control and maintenance of QoS. To perform QoS management, a number of QoS frameworks have been proposed such as [Zinky et al., 1997, Frølund and Koistinen, 1998b] and those surveyed in [Aurrecoechea et al., 1997]. In OMODIS, we are specifying a QoS framework designed to support the end-to-end view of QoS. The OMODIS QoS framework does not focus only on communication issues, but includes QoS management needed in database management systems. The OMODIS QoS framework performs QoS management activities in a uniform way so that system developers have a predefined set of services to use in their systems. This is discussed further in Chapter 6.

[1] In the OMODIS project, work is ongoing to specify a multimedia DBMS architecture that includes QoS-specific components.

2.6 Modelling Support

To be able to use the services a QoS framework provides, an appropriate interface is needed that gives an abstraction of the properties of the QoS framework so the application developers only need to care about the properties relevant for the applications. Many details regarding adaptation, negotiation, monitoring, etc., are not of importance to the application developers. For instance, the implementation details of most QoS mechanisms are unnecessary for application developers to be concerned with. Most QoS frameworks provide a declarative specification language that application developers use to specify the QoS properties of the system components and the requirements towards the QoS framework. Such languages are discussed in detail in Chapter 4 where one such language (CQML), defined as part of this work, is presented.

As mentioned in section 2.5, QoS is pervasive and any part of an application may need to have some QoS-related properties. Some of these properties may be given to the application after it has been implemented as a matter of configuration (e.g., by "throwing resources at it"), while other properties require a more careful approach in which conscious design decisions are called for during the development phase. Such design decisions can only be included in the design models if the necessary vocabulary is available to the modellers when producing the models. For instance, timeliness is an important category of QoS characteristics, and the modelling vocabulary for time-dependent applications needs to include the notion of time. Many software development methodologies already contain concepts with the expressive power necessary to specify QoS concepts, but may lack a coherent framework in which to position the different concepts in order to provide application developers with a useful approach to modelling QoS. In Chapter 5, we present an approach to incorporate QoS-aspects into software development by positioning the CQML language in object-oriented development in order to show traceability of QoS aspects from the application requirements to the properties of the underlying QoS framework.

2.7 Product Quality

As discussed in section 2.3, our primary concern is quality of service, not quality in general. However, we need to consider the product quality of the server providing the service. Next we present ISO Software Product Quality model that we use as a reference quality model.

2.7.1 ISO/IEC 9126

The ISO/IEC 9126 standard, Information Technology – Software Product Quality, was prepared by ISO/IEC Joint Technical Committee JTC 1 (Information Technology) Subcommittee SC 7 (Software Engineering). It consists of four parts: Quality model [ISO/IEC JTC1/SC7, 1999a], External metrics [ISO/IEC JTC1/SC7, 1999b], Internal metrics [ISO/IEC JTC1/SC7, 1999c], and Quality in use metrics [ISO/IEC JTC1/SC7, 2000a]. The first part presents a framework for quality specification, while the three other parts specifies how such specifications can be measured.

ISO/IEC 9126 is a framework for specification and evaluation of software product quality. As such, it provides a consistent terminology of different aspects of quality. It defines a two-part model for software product quality: internal and external quality, and quality in use. The first part identifies six quality characteristics that are further subdivided into subcharacteristics appropriate for internal and external quality. These characteristics are attributes of the software product that can be measured irrespective of context of use. Quality in use, on the other hand, represents the user's view of the quality of the system. It is measured in terms of results of using the system, as opposed to measurements of the software product itself. It identifies four characteristics appropriate to characterise quality in use.

2.7.1.1 Quality in Use

Quality in use is related to a specific context of use of the software product. It measures to what degree the users can reach their objective rather than measuring properties of the software product itself. In Figure 6, the quality model for quality in use is shown.

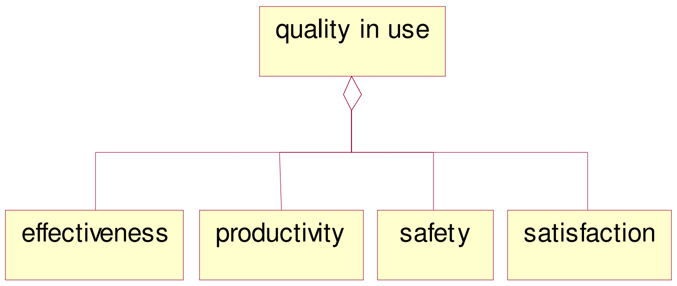


Figure 6: Quality model for quality in use

The terms in Figure 6 are defined as follows:

- quality in use – the capability of the software product to enable specified users to achieve specified goals with effectiveness, productivity, safety and satisfaction in specified contexts of use.
- effectiveness – the capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use.
- productivity – the capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in the specified context of use.
- safety – the capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use.
- satisfaction – the capability of the software product to satisfy users in a specified context of use.

2.7.1.2 External and Internal Quality

External and internal quality are specified using the same quality model; they differ only in their views. When specifying external quality, one focuses on the quality of the software product when it is executed, while internal quality focuses on parts of the system. In Figure 7, the quality model for external and internal quality is shown.

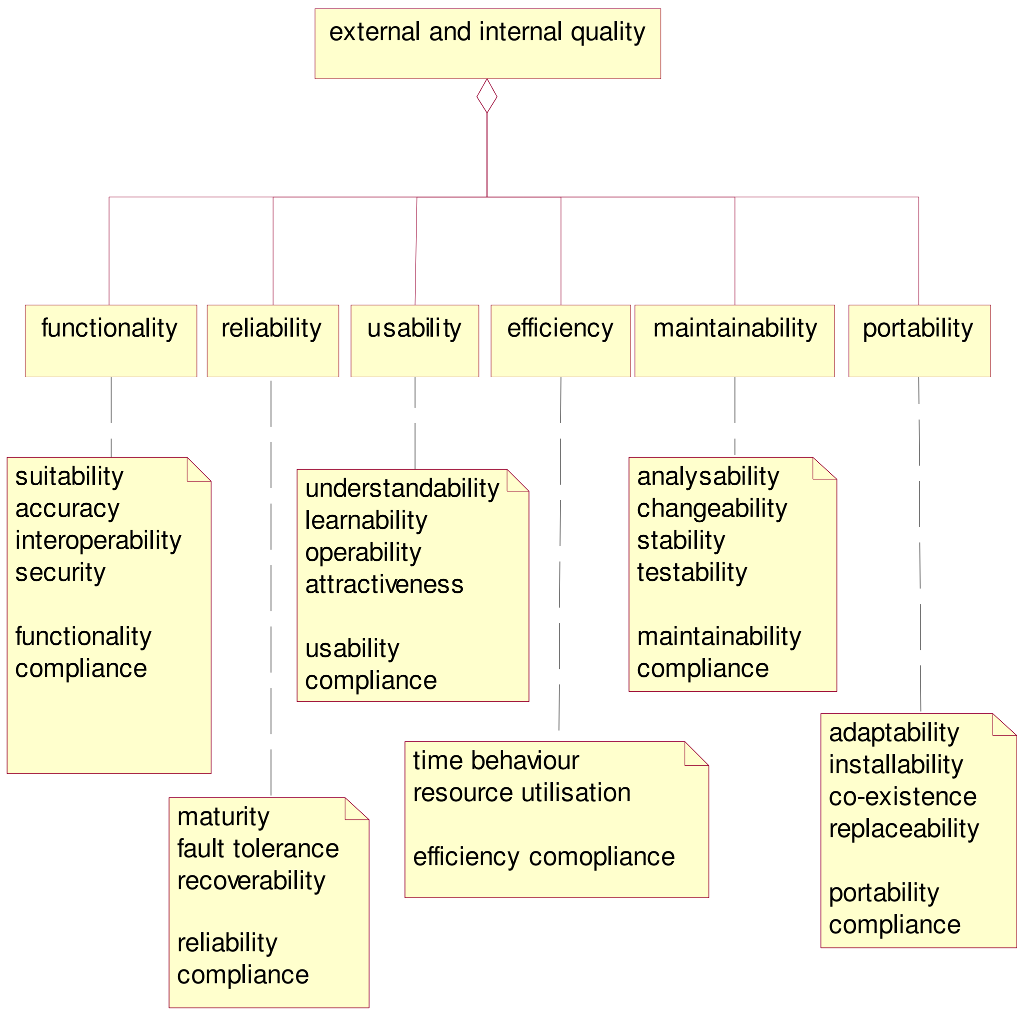


Figure 7: Quality model for external and internal quality

Each of the six characteristics has a number of subcharacteristics that further refine the categorisation of internal and external quality. The six characteristics are defined as shown below (also the subcharacteristics are defined in the quality model):

- functionality – the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.
- reliability – the capability of the software product to maintain specified level of performance when used under specified conditions.
- usability – the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.
- efficiency – the capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
- maintainability – the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

- portability – the capability of the software product to be transferred from one environment to another.

2.8 Summary

This chapter has provided a discussion of important concepts with respect to QoS. First a discussion of quantification concluded that even if vagueness inherently resides in any measurement, we can still choose a limit for characterisation of such measurements. Following this, a number of definitions of the term Quality of Service were provided and important concepts such as QoS characteristic, QoS statement, QoS requirement and QoS offer were defined. QoS can be specified at different levels of abstraction; this was the topic of section 2.4. Sections 2.5 and 2.6 gave brief introductions to the salient issues of infrastructure support for, and modelling of, QoS, respectively. Finally, section 2.7 provided an overview of the ISO Software Product Quality model.

Part II. Modelling

This part constitutes the main part of this thesis. First terminology is defined and different types of modelling are characterised before a chapter that defines a modelling language for QoS is included. Finally, a context for the modelling language for QoS is given that indicates how it can be integrated into software development.

Chapter 3 Modelling

Modelling is the activity of creating models. In order to make complex software systems, modelling is an essential activity. In this chapter we define terminology used in modelling, and characterise different types of modelling. The purpose of this chapter is to present a set of concepts that are essential for understanding the context of the subsequent chapters.

3.1 Terminology

From [Blum, 1994] we use the definition of method as a systematic process, technique or mode of inquiry that is used to aid in the creation of a satisfactory software product.

Using a method, one often produces models. A model is a representation of something, less complex than what it represents. It is a simplified description of a system used in explanations or calculations. In [Reenskaug et al., 1996], system is defined as a part of the real world that we choose to regard as a whole, separated from the rest of the world during some period of consideration. Models are used to describe various aspects of systems. A model is usually expressed in a specific notation. A notation, often called language, can be lexical or graphical.

Technique is often used for a specific construct supporting a method. For example, if a method tells how to create parallel processes, critical regions may be a technique used to achieve this. A technique sometimes uses an enabling mechanism. A mechanism is an underlying construct implementing the technique, often supported in the underlying infrastructure. If using critical regions is a technique, P and V operations on semaphores are mechanisms supporting this.

Methodology is defined in [Blum, 1994] as either a body of methods, rules, or postulates employed by a discipline or, more simply, the study of methods. In our case we use the word methodology as a body of methods. Examples of methods are Message Sequence Charts (MSC), Petri Nets, Entity Relationship Modeling (ERM), etc., none of which alone is meant to support all software development phases.

The methodologies are based on some underlying concepts. These concepts are often implicitly defined from the paradigm the methodology complies to. A paradigm represents a way of thinking and the pattern of basic primitives used to describe problems. Object-orientation is a paradigm that many methodologies follow nowadays.

Process is defined in [ISO, 1993] as a predetermined course of events defined by its purpose or by its effect, achieved under given conditions. More simply we can state that process means a sequence of actions leading to some result. The methods themselves often prescribe certain systematic processes, meaning they define which steps to take to develop a result. On a higher level, a process is also needed between the methods. This specifies in what turn to take what steps, meaning which method to employ and when to use it to get to an overall result. Hence, a methodology should prescribe both a set of methods, and how and when to use them.

A methodology can be depicted as in Figure 8.

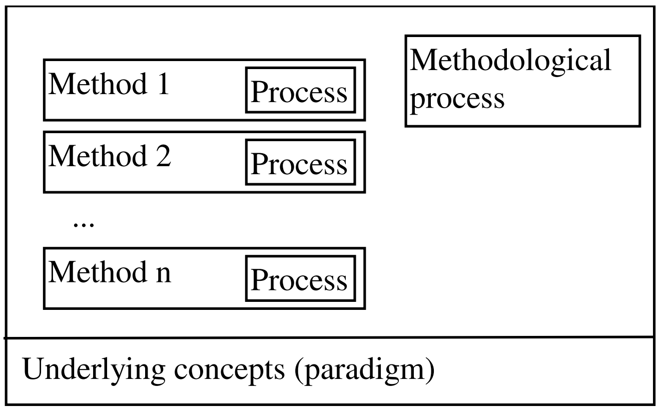


Figure 8: Methodology

3.2 Views, Abstraction and Meta-levels

The core of modelling is abstraction. To abstract is to describe only those issues that are important for a particular purpose. Abstraction helps us to illustrate specific aspects of a system uncluttered by irrelevant details. For a particular system, different abstractions are useful for different purposes. Abstractions of the same system that describe different aspects of it may be considered different views onto the same system. The different views are different models of the system. A paraphrase from [Ould, 1995] summarises this: Many models can be drawn from a system; all will be wrong, but some will be useful.

A model is always an abstraction of the system under consideration. However, models may be on different levels of abstraction. One model may be a refinement of another, meaning that it is representing the same aspects, but on a lower level of abstraction. In essence, it adds more detail. For a model to be a refinement of another, it needs to be conformant to the other. In other words, all properties of the abstract model should be preserved in the detailed model. To justify that one model is a refinement of another requires careful considerations. Actually, this is an area where most software modelling approaches are weak, and much could be learnt from work on formal methods where this has been addressed over a number of years. Most effort in defining refinement has been put into refinement of modelling elements within a model, e.g., subtype relationships, not on refinement of models as a whole.

There is a distinction between modelling the parts of a system individually or collectively. This template-instance dichotomy is essential in all modelling techniques. Even if the system consists of a number of instances, it is in most cases not practical to describe each instance. Instead, the features of objects are defined collectively for a set of similar objects. Such collective descriptions are termed templates and can be used to create instances.

If the system under consideration is a set of models, models can be created that describe essential aspects of the set of models. Such models are termed meta-models (models of models). A meta-model is itself a model that may be subject of modelling, such a model is termed a meta-meta-model. In [Atkinson, 1997], characteristics of different meta-levels are described. Note the difference between meta-levels and abstraction levels. Whereas a model on one abstraction level is a refinement of a model on a higher abstraction level, a model on one meta-level is an instance of a model of a higher meta-level.

3.3 Process and Life-cycle

The software life-cycle addresses development, use and maintenance of software systems. A software life-cycle model is a description of software evolution. Such models often include activities such as initiation of system development, requirement analysis, functional specification (design), architectural configuration, component implementation, software integration, testing, documentation, training, use and maintenance. The main objective of life-cycle models is to guide management of software development projects to reduce total costs throughout the lifetime of the system. Although life-cycle models also span use and maintenance, most descriptions relate to the development phase only. However, [Birrel and Ould, 1985] argue that maintenance almost always involves enhancements, and it can be handled as further development.

There are two approaches to life-cycle models: breadth-first and depth-first. Life-cycle models following a breadth-first approach address the entire scope of the software system and do a stepwise refinement from requirements to implemented system. Formal transformations may be used in these refinements. The original waterfall model [Benington, 1956, Royce, 1970] is breadth-first where all requirements are gathered before design starts. The waterfall model is sequential, where upon completion of one step one proceeds to the next without ever returning to a previous step. Breadth-first life-cycle models can also be iterative. Such models have a predefined sequence of phases, but have feedback loops between them so that needs for change discovered at a phase are used as input to an earlier phase, and the change is propagated forward through the phases. The iterative waterfall model is an example of this type of life-cycle model. Life-cycle models using a recursive approach re-apply the life-cycle model on results of the life-cycle. All recursive approaches are iterative, but not necessarily vice versa. The spiral model [Boehm, 1988] can be used recursively where each cycle may identify elements upon which the spiral model can be re-applied.

Depth-first approaches focus on parts of the system and develop these individually. Evolutionary development follows this strategy. One type of evolutionary development is exploratory programming where users and developers work closely together to develop the final system. The development starts with well-understood parts of the system, and as the understanding of other parts grows, these are developed and finally the system emerges. Extreme programming (often termed XP [Beck, 1999]) is a variant of this where focus is to relieve programmers from unnecessary activities prescribed by many current software methodologies. XP is one of several light-weight methodologies where the number of rules, practices and documents are kept to a minimum. Rapid prototyping is an alternative approach of evolutionary development where requirements are discovered by quick development of prototypes that users evaluate and provide feedback on. The prototypes are then often thrown away and improved versions are developed from scratch. In the end, the final system emerges.

Depth-first and breadth-first approaches are often combined, but on different levels. Rational Unified Process (RUP) [Jacobson et al., 1999] is an iterative and incremental approach. They prescribe to divide a system development project into smaller mini-projects. Each such mini-project is an iteration that results in an increment. In each iteration, the software development process is used (hence the term iteration). The result of one such instance of the process is an increment that represents a growth in the overall software system. The division of the main project into mini-projects follows a breadth-first strategy, but the development process used in each iteration may follow a depth-first strategy.

The distinction between software development methodologies and life-cycle models is not clear. As mentioned, life-cycle models tend, like software development methodologies, to focus on the software development phases only. However, software development methodologies also prescribe methods and techniques used in the individual phases of the development project. The objective of life-cycle models is to reduce the overall costs throughout the lifetime of the software system. This objective is equally important for software development methodologies and should be kept in mind when prescribing methods and techniques. The use of the prescribed methods and techniques should contribute to minimising the overall software costs, not only the development costs.

3.4 Summary

In this chapter, we have defined terminology used in modelling, and characterised different types of modelling. In section 3.1, we specified what we mean by terms such as model, method, process, and methodology, whereas in section 3.2, we discussed terms such as abstraction and abstraction levels, views, refinement, templates and instances, and meta-models. Finally, in section 3.3, we discuss software development processes and life-cycle processes, and identify two categories of such processes: depth-first and breadth-first.

Chapter 4 Component Quality Modelling Language

This chapter defines the Component Quality Modelling Language (CQML), a lexical language for specifying QoS. This chapter is a continuation and elaboration of ideas initially presented in a paper published and presented at IDMS'98 [Aagedal, 1998] (included in appendix I.D). We got important feedback on this paper from reviewers and during the workshop, and this feedback has been incorporated into CQML as seen fit. After [Aagedal, 1998] was published, other interesting results have been reported that address the same problem area. In this report, we have therefore incorporated novel ideas from other sources such as Quality Modeling Language (QML) [Frølund and Koistinen, 1998a] and Quality Objects (QuO) [Loyall et al., 1998] into a consistent modelling language, CQML.

We structure this chapter as follows. In section 4.1, the work is motivated and the context provided. Section 4.2 provides terminology used, while section 4.3 identifies and discusses requirements for a QoS specification language. Section 4.4 identifies existing approaches and evaluates them against the requirements identified in section 4.3. Section 4.5 presents the fundamental concepts and underlying semantics used, while in the main body of this chapter, section 4.6, CQML is defined. Section 4.7 provides a formalisation of the concepts and a discussion on substitutability and conformance between constructs in CQML. Section 4.8 presents an example of use. To provide a bridge to a QoS-enabled infrastructure, section 4.9 presents the compiler and the run-time representation of CQML. Appendix III provides the grammar of CQML, while appendix I gives the IDL for the run-time representation of CQML.

4.1 Introduction

One tool in assisting the development of distributed applications is interface definition languages. Such languages facilitate implementation-independent abstractions of functional properties of systems or parts thereof. Popular interface definition languages like CORBA IDL [OMG, 1996] and Microsoft IDL [The Open Group, 1997] support specification of functional properties as attributes and operational signatures, but lack any support for specification of QoS.

International Organization for Standardization (ISO), Telecommunications Information Networking Architecture Consortium (TINA-C) and Object Management Group (OMG) are all working to incorporate QoS into their architectures [ISO/IEC JTC1/SC21/WG7, 1997, TINA-C, 1994, Sluman et al., 1997]. However, there is yet no way of precise specification of QoS in these architectures. Here we propose a language, CQML (the Component Quality Modelling Language), in which to state precise specification of QoS.

4.2 QoS Terminology

We base the terminology regarding QoS on the ISO QoS Framework [ISO/IEC JTC1/SC21, 1995b] and the ongoing work on QoS in Open Distributed Processing (ODP) [ISO/IEC JTC1/SC21, 1998]. This section is an elaboration of parts of section 2.3 relevant for specification of QoS.

4.2.1 Basic Concepts

According to the ISO QoS Framework, QoS characteristic is the most fundamental term to express QoS. A *QoS characteristic* represents some aspect of the QoS of a system, service or resource that can be identified and quantified. It represents the true underlying state of affairs (the actual, performance-oriented description of the behaviour), as opposed to any measurement (see QoS observation below). Examples of QoS characteristics are time delay, throughput, availability, and precedence. QoS characteristics are to be distinguished from

QoS parameters. A *QoS parameter* is any value related to QoS that is conveyed between entities.

In the ISO QoS Framework, a number of QoS characteristics are defined. However, there is no exhaustive list of QoS characteristics, the list in the ISO QoS Framework reflects only an agreed set of common characteristics within that framework focussing on communication. [ISO/IEC JTC1/SC7, 1999a] presents a quality model for software products that includes another set of characteristics spanning a broader area than communication only.

QoS characteristics can be defined in terms of other characteristics in two ways: by specialisation and by derivation. Specialisation makes an abstract characteristic more concrete by adding details. Time delay may for instance be specialised into transit delay that specifies the delay in some transmission. This may be further specialised to take into account the specific points between which the delay is defined, to what type of information and to what type of connection it applies. Some QoS characteristics can be defined as functions of others; these are derived QoS characteristics. Statistical characteristics are one important group of derived QoS characteristic. These include maximum, minimum, range, mean, variance, standard deviation, n-percentile, and statistical moments, all defined in the ISO QoS Framework. Application of these derivations can lead to QoS characteristics such as mean transit delay, variance of transit delay, etc. Other derived QoS characteristics include some that are functions of more than one characteristic. Availability is one such that can be defined as a function of reliability and maintainability (e.g., how fast the system can be repaired if broken).

QoS characteristics can be grouped into QoS categories. A *QoS category* represents a type of user or application requirement. Different types of requirements lead to different QoS categories (sets of QoS characteristics). The ISO QoS Framework defines some fundamental QoS categories, and this (non-exhaustive) list includes categories such as security, safety, timeliness, capacity, integrity, coherence and reliability.

QoS characteristics are quantified within a *domain* of values. A specification of a QoS characteristic is a constraint over its domain. Some characteristics are quantified over a numeric domain (e.g., the non-negative integers) while others are quantified over an enumerated or a set domain. The theory of software metrics [Fenton, 1991] defines five scale types for transformations from entities in an empirical relation system to a numerical relation system. The scale types are: nominal, ordinal, interval, ratio and absolute. A *nominal scale* is any one-to-one mapping between the elements and scale values. Labelled elements have nominal scales. In an *ordinal scale*, the ordering of the elements is preserved in the mapping. Enumerated elements have an ordinal scale. An *interval scale* preserves the distance between the elements in the mapping while a *ratio scale* adds a zero relation. Temperature is often quantified using an interval scale. An increase from 10 to 20 degrees does not indicate that the heat has doubled, only that the number of degrees has doubled. Timing is an example where a ratio scale is used. Since it has a zero relation, an increase from 10 to 20 seconds means that the measured time has doubled. Finally, an *absolute scale* is restricted to the identity mapping. “Number of failures” is an example where such a scale is used. The scale type used for a QoS characteristic, if any, restricts applicable derivations. For instance, it makes no sense to compute the mean when using an ordinal or a nominal scale.

Expressing the total QoS in terms of a set of QoS characteristics facilitates reasoning about QoS. Note that specifying QoS characteristics is relevant for interfaces, operations, attributes, parameters and operation results. From the QoS in ODP working draft [ISO/IEC JTC1/SC21, 1998], there are five categories of *QoS statement* used when describing QoS (or its constituent QoS characteristics), and subsequently in building distributed systems:

i) *QoS requirements*, which state QoS constraints that users of the system or its components have on the system or its components. They can be expressed using the concept of QoS relation. QoS relation is the most fundamental concept for describing non-functional aspects of systems. It will be described below.

ii) *QoS capabilities*, which state the actual abilities provided by the components or configuration of components in the system with regard to QoS. QoS capabilities can be expressed using the notion of QoS relation.

iii) *QoS offers*, which describe advertised QoS. While QoS capabilities describe actual capabilities of the components, QoS offers are introduced to take into account uncertainty of the knowledge of the actual QoS capabilities of objects. This uncertainty can lead to some discrepancy between an object's QoS capability and its advertised offer. QoS offers can also reflect the fact that a total QoS capability can be partitioned so that only subsets are available as advertised offers. QoS offers can be expressed using QoS relations.

iv) *QoS contracts*, which are the results of agreement processes. They define the sets of capabilities that objects in a configuration have agreed to provide to each other. They can also be expressed using the QoS relation concept. They can either be agreed by dynamic negotiation mechanisms or be implicitly incorporated into the design of systems.

v) *QoS observations*, which express the values of QoS characteristics as observed using measurement functions.

A *QoS relation* specifies the mutual obligation of an object and its environment with respect to QoS. The QoS relation concept is the most primitive concept for specifying QoS aspects in ODP systems. This concept is introduced to model the fact that, in general, QoS provided by an object can also be a function of how the environment performs. For example, the QoS of a multimedia presentation object can depend on the QoS of a video storage service and an associated audio storage service. A QoS relation consists of an expectation part, which describes QoS expectations of an object from its environment, and an obligation part, which describes the object's QoS provision, provided the environment performs according to the expectations. More formally, for an object `O`, this is expressed through the relation

$$\texttt{Exp(O)} \mapsto \texttt{Obl(O)}$$

also known as an assumption/guarantee formulae [Jones, 1983], where '$\mapsto$'[2] is a form of logical implication that can be read as 'constraint `Obl(O)` is met as long as constraint `Exp(O)` is met'. '$\mapsto$' is used to specify open systems, i.e., systems that interact with an unspecified environment, and is a stronger version of logical implication '$\Rightarrow$' than the traditional version. The restriction of the implication, presented in [Abadi and Lamport, 1994], is that the implication should be valid for all possible prefix behaviours of `Exp(O)`, i.e., at any point in time, it is not so that `Exp(O)` is valid while `Obl(O)` is not valid (this property is called μ–receptiveness in [Abadi and Lamport, 1994]). Also, `Exp(O)` and `Obl(O)` should be safety properties. Safety properties are refutable in finite time, i.e., it is a property that is satisfied for an infinite history iff it is satisfied for every prefix of the history [Alpern and Schneider, 1985].

With this restricted version of implication, QoS relations can be composed to be applicable to a composition of objects, see subsubsection 4.6.3.5.

Note the difference between QoS require ments and QoS expectations; QoS requirements are constraints someone in the environment of an object has on the object while QoS expectations are constraints an object has on objects in its environment. In a given object configuration, the QoS requirements for an object depend on all QoS expectations for all other objects that are to be met by the object in question. QoS relations consist of expectations and obligations, hence they are focussed on one object only; an object need not be aware of other objects in its environment beyond knowing that objects exist that can meet its expectations.

[2] $\mapsto$ is represented in [Abadi and Lamport, 1994] using a symbol similar to $\pm\triangleright$

To illustrate how QoS relations can be used to specify the different QoS statements, we use the simple configuration in Figure 9.

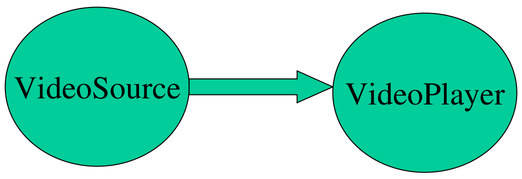


Figure 9: Video player example

In Figure 9, the video player gets information from a video source that is shows to end-users. It offers "good" video quality if it gets video information with at least a frame-rate of 25 per second. The video source does not depend on any objects in its environment, but offers to deliver at least 30 frames per second of video information. The QoS offer of the video source can be specified as:

```
TRUE ↦ framerate >= 30
```

The QoS relation of the video player can be specified as:

```
framerate >= 25 ↦ good
```

Given that the video source is to provide the service to the video player, the QoS requirement of the video source is to meet the expectation part of the QoS relation of the video player (i.e., the video player requires this from the video source), that is `framerate >= 25`. If the system designer decides that the video stream is to have a frame-rate of 26 frames per second (different from both the QoS requirement and the QoS offer), it can be specified as another offer from the video source with the new value.

QoS capabilities are specified in the same manner as QoS offers.

4.2.2 Management

Systems may manage QoS in different ways. At one extreme, QoS requirements can be met statically during design and implementation by proper design and configuration choices (such as scheduling rules, network bandwidth allocation, etc.). This will give a well-defined behaviour, but without any flexibility. At the other end is the dynamic approach that lets the systems negotiate at run-time the restrictions of the QoS characteristics they need for their activities. This approach often involves an adaptive aspect by having monitors and corrective operations to be taken when the QoS level drops below a certain level. This approach is very flexible as the QoS policies can be changed at run-time, but the behaviour is not well defined and the performance may be degraded if costly adaptation schemes are used.

From the ISO QoS Framework, QoS management activities can be performed at different stages of a system activity:

- a priori – appropriate functionality built into the system at design time, e.g., by dedicating resources, systems sizing, etc.
- before initiation – QoS requirements are conveyed to system components before initiation, for instance so that appropriate resources can be reserved.
- at the initiation – QoS requirements are negotiated between provider and consumer.
- during the activity – the environment may change so that re-negotiation of the initial QoS contracts may be required.

- historically – statistical analysis of the activity, such as trend analysis, contract evaluation, etc.

Note that QoS management is performed for a system activity, not only for the overall system execution.

Also, different classes of QoS can be specified that influence what types of QoS management is needed. The following types of commitment of the service provider are usually distinguished:

- Best-effort QoS: The system does its best to meet the required level of QoS, but it may fail to do so under certain circumstances.
  - Threshold QoS is a type of best-effort QoS where the user is warned if the QoS is below the threshold level.
  - Compulsory QoS is another type of best-effort QoS where the system aborts if the QoS is below the compulsory level.
- Guaranteed QoS: The system guarantees that the specified QoS will be met if the environment satisfies certain requirements. Guaranteed QoS always requires that the environment meet certain conditions; a sufficiently hostile environment can always ruin system performance. Deterministic QoS is when there are no variations of the anticipated and provided QoS. QoS can also be guaranteed according to a statistical distribution. Also, maximal quality QoS can be specified. The guarantee is then not on a lower level of QoS, but rather an upper limit, for instance to avoid saturation of the client.

Many applications will need a combination of these approaches.

4.3 Requirements

A modelling language to support specification of QoS may be useful in a number of phases in the software life-cycle, and may be used for different purposes by a number of actors in system development. The analyst may use it to specify non-functional (sometimes termed extra-functional) user requirements. The system developer may use it to specify QoS offers and QoS expectations of individual system components. The QoS framework developer may use the language to specify the QoS properties of the framework components. The QoS framework itself may use the specifications from the system developers to represent the QoS properties of the components, and it may use this information to perform QoS management activities such as negotiation, monitoring, etc.

A major distinction exists between languages that support definition of behaviour, e.g., that enable specification of how required QoS mechanisms work, and those that support specification of system properties where only the effect of the behaviour is defined, irrespective of how it is implemented. The language presented herein falls into the second category. However, even if definition of system behaviour is not supported, the language facilitates QoS management by providing the system designers with a tool to specify the QoS properties of the individual components. This information is in many cases crucial to perform QoS management.

We identify the following requirements for a modelling language to support QoS. We also provide their rationale:

1. Generality – It should be generic, not tailored towards specification of a particular QoS category or a restricted set of QoS characteristics. This ensures that the language can be part of a general methodology and be used by generic modelling tools.

2. Life-cycle support– It should support specification of all aspects of QoS during the entire software development process. In particular, it should be possible to specify both quality in use and external quality. This enables specification of QoS in both the problem and solution domain of software development.

3. IDL integration – It should support integration with system specifications specified using well-known interface definition languages. This reduces introduction costs since well-known languages can still be used and existing applications left unchanged.
4. Platform independence – It should be platform-independent to avoid focussing on specific features available in one platform only. This also extends the scope of usage of the language.
5. RM-ODP compliance –It should be compliant with RM-ODP. RM-ODP offers a sound starting point for technology-neutral specification of architectures for open distributed systems and has a high influence on standards from organisations like TINA-C and OMG.
6. Object-oriented – It should integrate seamlessly with object-oriented concepts. Object-orientation is a prevailing paradigm of software development and a language should support the basic concepts in this paradigm. However, the previous requirement subsumes this since RM-ODP is a reference model within the object-oriented paradigm.
7. UML integration – It should integrate seamlessly with UML. UML is a popular graphical modelling language in widespread use across many domains, and a definition language that supports QoS should support use of UML in the software development process.
8. Separation – The specification of the QoS a component offers should be separated from the specification of its functional properties. This is a separation of concerns between behaviour (functional properties) and constraints on it (the QoS it offers), and is typical of dual language techniques such as TTMs/RTTL [Ostroff, 1992] and Esterel/QL [Stefani, 1993], and is also argued in [Blair et al., 1997] (together with additional separation of concerns between specification of different time-related issues).
9. Syntactic separation – QoS specification should be syntactically separate from interface definitions of the functionality. This facilitates that a functional interface can be implemented by many components, possibly with different QoS. This also enables existing tools that process interface definitions to be used unchanged. Finally, this requirement facilitates the use of different interface definition languages since QoS specifications are not inserted into interface specifications.

   Note the subtle difference between this and the previous requirement; whereas the previous requirement was that specification of behaviour should be separated from specification of constraints on that behaviour (in particular, constraints defining QoS), this requirement is that any specification related to QoS (including specification of behaviour such as negotiation capabilities) should be separated from the functional specification. Meeting both requirements means that functional and QoS-related behavioural specifications are separated, in addition to their separation from QoS requirements.

   Both types of separations have the negative effect that developers need to know more than one specification in order to develop the service. However, we believe this disadvantage is outweighed by the benefits of separating the specification types.
10. Predefined adaptation – One should be able to specify predefined adaptation schemes using the language. This approach to adaptation requires that one is able to specify a number of QoS offers that a component can provide depending on the prevailing environmental conditions. While supporting adaptation, predefined adaptation schemes make it possible for the system to prepare itself for a set of possible configurations.
11. Negotiation – The language should support features needed for negotiation between client and server in order to establish QoS contracts. It may not always be possible to establish a QoS contract without negotiation that involves the exchange of offers and counter-offers. However, for a service provider to be able to produce offers of increasing value to the client so that the negotiation process may converge, the service provider must be able to evaluate the client value of the offers and this must be supported in the language. This

also facilitates unforeseen adaptation since a QoS violation may trigger re-negotiation, not only transition between predefined QoS offers as the previous requirement enables.

12. Adaptive behaviour – One should be able to specify the triggering of behaviour when adaptation occurs. If the capabilities of a service provider are reduced, clients using this service may want to initiate compensating behaviour and it should be possible to specify this.
13. Composition – QoS statements should be composable, i.e., it should be possible to specify QoS at a fine-grained level and compose such specifications into higher-level specifications. Inversely, a high-level QoS statement should be decomposable such that fine-grained QoS statements can be refined. This supports the well-known software engineering principle of divide-and-conquer by making possible decomposition of component specifications that include QoS into finer-grained component specifications that include corresponding finer-grained QoS specifications. Both top-down and bottom-up approaches of software engineering are supported by this requirement. It should be noted that the refinement relationship between two levels of QoS statement must ensure consistency between the two levels. The process of composition and decomposition should be carried out in such a manner as to guarantee this.
14. Refinement – QoS specifications should be able to be refined so that new QoS specifications can be based on existing ones. This is an application of the inheritance technique in object-orientation that facilitates reuse and sharing.
15. Typed – The language should be typed to facilitate conformance checking when negotiating QoS contracts, e.g., by using a trading mechanism.
16. Precision – The language should support precise definition of the specified QoS. Precision reduces the number of potential misunderstandings between developers as the concepts defined have fewer aspects left to individual interpretations. Also, precise specifications facilitate generation of parts of the application as requirement 21 addresses.

The following requirements focus on the use of the language:

17. Interface annotation – The modeller should be able to annotate interfaces and parts thereof with QoS statements. This allows implementation-independent specification of QoS such that any component implementing this interface has both functional and non-functional properties to fulfil.
18. Component annotation – The modeller should be able to annotate components with QoS statements. This allows complete specification of components, including their non-functional properties.
19. Combinations – The modeller should be able to represent different combinations of components with QoS properties, i.e., serial, parallel and hierarchical configurations. For instance, by composing two channels in series, the smaller of the two throughputs is the resultant throughput. If they are composed in parallel, the throughputs can be added. Hierarchical combinations are addressed by requirement 13.
20. Expectations and offers – The modeller should be able to specify both what a client needs of QoS and what the service provides. The QoS requirement of the client can then be matched by a QoS offer by the service so that a QoS contract can be agreed upon. Also, by specifying both what a component offers and what it requires, we specify the complete QoS characteristics of the component and it can be reused in different contexts.
21. Code generation – The system developer should be able, from the QoS specifications, to generate parts of the system for monitoring and management of the specified QoS. Many of the parts needed for QoS management can be defined generically and parameterised so that when used, it is only a parameterisation of a generic implementation. This approach relieves system developers from the mundane tasks of re-implementing essentially the same functionality many times, also reducing the number of error sources.

The following focuses on the requirements of an infrastructure that uses the language to facilitate QoS management:

22. Agreement process – Both static and dynamic ways of agreeing on QoS contracts should be supported. The static approach requires that one can specify QoS contracts while designing the system, while the dynamic approach requires that QoS offers and QoS expectations should be available at run-time so that matching of these can be performed. The dynamic approach may also involve run-time generation of new QoS offers and QoS expectations if no appropriate ones are specified at design time.
23. Adaptation – The infrastructure should support application adaptation whatever the cause. This means that one should be able to specify ranges of QoS offered or required and priorities of these so that changing environmental conditions can lead to the best possible new QoS contracts within the valid range.
24. Observable – The QoS of the modelled components should be observable so that monitoring applications can reason about QoS observations. To allow application adaptation, monitoring applications need to report QoS observations so that corrective actions can be taken if the QoS value is below a certain limit specified in the QoS contract.
25. Comparison – The supporting infrastructure should be able to represent comparisons of QoS values gained at different locations. Transfer delay is one such QoS where two QoS observations at different locations can be compared and reflected into one QoS characteristic.

4.4 Existing Approaches

A number of languages support QoS specifications in specific QoS categories like timeliness or reliability, while others address generic specification of QoS covering all QoS categories. In this section we give an overview of related work both on tailored and general-purpose languages, and provide a qualitative evaluation of these approaches with respect to how they meet the requirements identified in 4.3.

4.4.1 Formal Methods

4.4.1.1 Overview

When developing software systems, one must specify in detail the behaviour of the system; most commonly in a programming language like C++ or Java. Whereas such programming languages usually are only defined informally, formal methods include alternative specification techniques based on sound mathematical foundations. Because of the mathematical foundation, formal methods can be used to reason about the properties of software systems with different degrees of rigor. Firstly, formal methods can be used to specify properties of a system. Such a specification may include assumptions about the environment in which the system will operate, guarantees of system behaviour in such an environment and a design of the system to meet these guarantees. Secondly, formal methods can be used to prove (using theorem proving or model checking) that certain properties are valid or that one specification is a refinement of another. Thirdly, formal methods may include mechanical theorem provers that partly automate proof discovery and validity checking.

In [Jones, 1992], the author traces the history of research on reasoning about programs back to the pioneering work by Herman H Goldstine and John von Neumann in 1947 [Goldstine, 1947] and Alan Turing in 1949 [Turing, 1949]. The seminal work of Dijkstra and Hoare on program verification is considered by many the seed of formal methods [Dijkstra, 1968, Hoare, 1969]. A large number of formal methods exist with different underlying mathematical foundations. A major distinction is made between model-based and algebraic approaches. Model-based approaches such as Z [Spivey, 1988] and VDM [Jones, 1986] are based on set theory and logic, whereas algebraic approaches such as LARCH [Guttag and Horning, 1993] and ACT ONE [Ehrig and Mahr, 1985] are based on equational logic. However, these approaches focus on specifying sequential systems. The problem of concurrency is addressed by process algebras such as CCS [Milner, 1989] and CSP [Hoare, 1985] where message-passing is central, and by model-based approaches such as UNITY [Chandy and Misra, 1988] and TLA [Lamport, 1994] where states and state changes are of primary concern. Hybrid approaches such as LOTOS [Brinksma et al., 1987] combine these approaches.

4.4.1.2 Existing Approaches for QoS Specification

Since formal methods can be used to specify the behaviour of a system, such methods can also be used to specify QoS. Most existing approaches that address this focus on real-time issues. [Heitmeyer and Mandrioli, 1996] includes a survey of formal methods for real-time systems. Focussing on distributed systems only, [Dietrich and Hubaux, 1999] is a survey of formal methods for development of communication services. In [Bowman et al., 1995] and [Stølen, 1998], formal methods to specify open distributed systems are evaluated.

SDL [Ellsberger et al., 1997] is the leading language for specification within the telecommunication domain. Even if SDL is a language with a mathematical foundation, it has a graphical notation that makes it appealing to software developers. SDL is a state machine-based approach following in the footsteps of Statecharts [Harel, 1987], where processes are represented as state-machines with asynchronous message-passing for communication between them. SDL supports only time-outs, not bounded time requirements [Leue, 1995], but a temporal logic extension proposed by Leue remedies this.

The real-time logic QTL (Quality of Service Temporal Logic) [Blair et al., 1993, Blair and Stefani, 1997] is one part of a dual language approach to specify timing requirements and performance assumptions; LOTOS is used to specify functional behaviour. QTL is event-based with linear real-time, employing trace interleaving semantics. SQTL [Lakas et al., 1996] is an extension that includes stochastic constructs.

An influential work in formal specification of QoS is the temporal logic of actions (TLA). It is based on temporal logic as presented in [Pnueli, 1977] and was first presented in [Lamport, 1994]. TLA is a state-based temporal logic in which assumption/guarantee specifications can be made to model how a system operates if the environment does what it is supposed to do. TLA is the underlying formalism of QoS in ODP.

MT-LOTOS [Février et al., 1997] extends LOTOS with real-time and gate passing. The real-time extension allows quantification of time while the gate passing extension allows adaptation through dynamic reconfiguration. With these extensions to LOTOS, MT-LOTOS is useful to specify the behaviour of objects that meet some QoS requirements. However, to specify such requirements, a complementary notation is needed. TLA and Calculus for Object Contracts (COC) [Février et al., 1997] are mentioned as candidates for such a notation. COC is an action-based calculus in which a contract is defined as an observer of a set of interfaces that identifies and reports errors. Both approaches can be used to specify QoS requirements.

[Ren et al., 1997] proposes an extension to an actor-based formalism to include specification of time-related QoS attributes. Their approach is to include special actors called QoS synchronisers to manage QoS constraints on the media actors.

At University of Kent and Lancaster University, work is ongoing that targets to investigate formal methods for specification and validation of QoS properties [Waddington and Hutchison, 1998, Blair et al., 2000].

4.4.1.3 *Conclusion*

Languages exist, both formal and informal, that are tailored for specification of QoS within certain QoS categories. Within application domains such as multimedia and telecommunication, QoS is an important part of most applications and efforts have been made to facilitate specification of QoS within these contexts. These approaches focus mainly on temporal aspects of QoS. For instance, in [Aagesen, 1997], where state-of-the-art QoS frameworks for distributed systems in the telecommunication context are discussed, QoS is defined as "*a measure of the relative frequency of specific events or duration of times between specific events within a service, used as a quality criteria for proper service functioning*". This view of QoS excludes non-temporal QoS characteristics such as the failure masking property of a service (e.g., halt, initial state, rolled back), transactional properties (e.g., serialisation, atomicity), information quality (e.g., accuracy, validity), subjective user satisfaction, etc. Using the term service as the behaviour of some functional capability, one may argue that such characteristics are not related to the behaviour of the system providing the service. However, the capability of a system to satisfy its users may depend on such non-temporal characteristics and is therefore relevant to include when specifying QoS. As the definition in [Aagesen, 1997] indicates, focus on temporal aspects only is the prevailing view of QoS in these domains. However, even if this is a restricted view, temporal QoS characteristics constitute a significant class of QoS characteristics and the approaches that address this subclass are important to be aware of when defining a general-purpose language for QoS specification.

Despite experiences showing that formal methods are applicable to industrial software development (in particular of safety- and security-critical systems) [Gerhart et al., 1994], formal methods have not been widely accepted in software engineering [Clarke and Wing, 1996]. Reasons for this are, according to Clarke and Wing, that notations are often obscure, tool support is in many cases missing, and perhaps most importantly, most techniques do not scale. However, even if formal methods are considered unsuitable for use by most application developers, they are useful as foundations for more high-level specification languages. For instance, the assumption/guarantee (also called assumption/commitment) paradigm is useful as a formalisation of the software engineering principles of separation of concern and contract-based development. Actually, most of the principles in formal methods have their corresponding concepts in the more pragmatic world of software engineering and in widely used programming environments. E.g., the assumption/guarantee paradigm is equivalent to outgoing (assumption) and incoming (guarantee) interfaces in the Java and CORBA component models (EJB [Shannon, 1999] and CCM [OMG, 1999a], respectively). Lately, formal techniques such as OCL have been directly incorporated into informal modelling approaches, bridging the gap between formal methods and software engineering even further.

For specification of QoS, the inclusion of time in the formal method is essential. TLA and QTL/SQTL are both useful approaches for this. Also, the principle of separation of concerns is important. This ensures that functional behaviour can be specified and verified independent of the QoS properties of the system. It is important to note that to support specification of QoS, many formal methods are too expressive and only parts of them are needed. When specifying QoS, it is only properties that a system must possess that need to be defined; the design of a system that meets these properties is outside the scope of QoS specification.

4.4.2 Architecture Description Languages

Architecture Description Languages (ADLs) have been proposed to support development of distributed systems, but there is little consensus on what an ADL actually is. [Medvidovic and Taylor, 1997] presents a classification and comparison framework for ADLs wherein the most prominent ADLs are analysed. Medvidovic and Taylor define an ADL to include capabilities to explicitly model components, connectors and their configurations. In this context, a component is a unit of computation or a data store. A component has a set of interfaces that specifies the services (messages, operations and variables) it provides. An ADL may also include facilities to specify what services a component requires. Connectors model interactions between components. Configurations are connected graphs of components and connectors that describe architectural structure. There is, however, little support for QoS in ADLs. Rapide [Luckham et al., 1995] allows modelling of timing constraints, but ADLs are generally more focussed on modelling the functional aspects of configurations of system components than the non-functional aspects.

Since ADLs do not support QoS specification, a QoS specification language may complement the ADLs to better support specification of distributed systems. An ADL integrated with a QoS specification language will provide facilities for system developers to specify both functional and non-functional aspects of systems. A system can be specified without considering QoS aspects, but QoS aspects cannot be specified without relating them to service providers. Hence, an ADL does not need to be integrated with a QoS specification language, but a QoS specification language needs to be integrated with a language that provides facilities for specification of service providers. Such a dual language approach constitutes a clear separation of concern. With the advent of component models such as Enterprise Java Beans and CORBA Component, their component definition and configuration languages may also meet these requirements.

4.4.3 SMIL

SMIL (Synchronized Multimedia Integration Language) [Hoschka, 1998] is a markup language for multimedia on the Internet, defined as an XML DTD (eXtensible Markup Language Document Type Definition). It enables the description of the temporal behaviour of a presentation by providing a time line for co-ordinating the display of multimedia objects. Multimedia objects can either be presented sequentially (using the "`seq`"-element) or in parallel (using the "`par`"-element), and synchronisation between them can be specified either based on absolute time or events. Adaptation is supported by the "`switch`"-element that evaluates properties of the host system and displays the multimedia objects accordingly. Properties that can be tested in a switch-element include system-language, system-bitrate, system-screen-size, system-screen-depth, etc.

SMIL is not a general-purpose QoS specification language. In fact, it is restricted to specification of multimedia presentations. However, SMIL facilitates adaptation, albeit restricted to some properties, by providing the host system possibilities to examine its resource availability and, based on this, decide how to present the multimedia objects.

4.4.4 TINA ODL

TINA ODL [TINA-C, 1996] is a superset of CORBA IDL that enables the specification of computational objects with operational and stream interfaces. In contrast to CORBA IDL, TINA ODL supports specification of QoS. The approach to QoS specification is the simplest possible; QoS is specified using name-value pairs directly related to a stream or an operation. The particular QoS semantics to be supported has not been finalised, leaving the interpretation of what a name-value pair means to the developers (e.g., best-effort, average, guaranteed, etc.). Also, since a QoS specification in TINA ODL is directly connected to a stream or an operation, it is not possible to associate different QoS specifications with the same interface, preventing different implementations of the same interface with different QoS. This can, however, be achieved by defining separate derived interfaces that add different QoS specifications.

4.4.5 MAQS QIDL

Becker and Geihs have in a number of publications presented their QoS management architecture MAQS (Management for Adaptive QoS-enabled Services) [Becker and Geihs, 1997, Becker and Geihs, 1998, Becker and Geihs, 1999, Becker et al., 1999]. MAQS includes QIDL, an extension to OMG IDL that supports specification of QoS. QIDL extends IDL by providing the possibility to specify QoS interfaces and to assign these to functional interfaces. In a QoS interface, different QoS characteristics are defined using regular IDL types and operations can be specified that provide QoS-related operations to clients. Such QoS-related operations may, for instance, be used to convey QoS parameters between client and server, or to change QoS management policies.

The simplicity of the QIDL approach is appealing, in particular the use of a predefined general-purpose type system to define QoS characteristics and the fact that QIDL supports declarative specifications only, not mixing in implementation details that belong to another level of abstraction. However, the approach has some disadvantages as discussed next.

QIDL is tied to CORBA IDL, disqualifying other interface definition languages. Not all application developers will appreciate such a restriction, especially if they are skilled in a language other than CORBA IDL. Albeit, since QIDL essentially groups attributes, marking them as QoS characteristics, and associates these attributes with interfaces, QIDL can presumably easily be mapped into other interface definition languages since such languages in most cases support the same basic concepts (interfaces and attributes). However, such an approach will lead to different QIDL dialects, undermining reuse and portability.

The ability of two components to implement the same interface, but with different QoS, is not supported. Using QIDL, separate interfaces must be defined if components implementing the same functionality support different QoS. Using inheritance in IDL, this can be achieved by defining separate derived interfaces only differing in their QoS specification.

The QoS interface in QIDL consists only of one set of operations. This follows the tradition of CORBA where objects only implement one interface. In infrastructures such as COM+ and Java, on the other hand, objects may implement several interfaces. From the perspective of both software engineering and object versioning, it is sometimes useful to separate what services to offer to different sets of clients, but the current version of CORBA IDL does not support this. QIDL adopts this deficiency by only having one QoS interface that clients can use to access QoS information. It would be useful to distinguish between QoS operations used during service operation, QoS operations used during negotiation and QoS operations used during QoS management for components in a QoS framework, e.g., a monitor. From a software engineering point of view, different interfaces identify different roles facilitating separation of concern, while from a robustness point of view, different interfaces enable each interface to be changed without affecting clients only interested in the other interfaces.

QIDL only allows QoS characteristics to be assigned to interfaces, not any of its constituent parts. Although this reduces complexity by avoiding potential conflicts between QoS characteristics assigned to the interface and QoS characteristics assigned to a part of that interface, it restricts the precision of QoS specification to the interface level. At times, it may be useful to specify characteristics of operations and even parameters, for instance the validity or accuracy of an information object, but QIDL does not support this.

The vocabulary of QIDL to define QoS characteristics is restricted to CORBA IDL types. This excludes definition of user-defined types such as numeric ranges or sets. As the previous issue, this also limits precision in QoS specifications.

QIDL has no direct support for adaptation. The approach taken in MAQS is to predefine interfaces to use during negotiation, that QoS-enabled clients and servers must support. In this way, MAQS supports adaptation by re-negotiation. However, it is not possible for a service provider to decide during negotiation what QoS offer best suits the client it is negotiating with other than by implicit knowledge. Nevertheless, as already indicated in [Becker and Geihs, 1999], a worth function can easily be added in the client QoS interface (`QoSCLFW` in MAQS), enabling dynamic adaptation without prior knowledge of client preferences. The possibility to specify statically different levels of QoS that can be provided, on the other hand, cannot be easily included in QIDL. This approach to adaptation may relieve the system from engaging in costly negotiations, but its inclusion in QIDL would require explicit precedence relations between different instances of QoS interfaces and the triggering of transition behaviour when a QoS interface instance is invalidated.

In QIDL, QoS interfaces are defined using CORBA IDL. Besides constant definitions, CORBA IDL is used to define types and does not include values. This means that QoS interfaces in QIDL are on the type level and no values are available to restrict instances of these types. QoS contracts between individual clients and servers are, on the other hand, instances, where each type of QoS characteristic defined in the QIDL QoS interface must be given a value or left undefined. This is not possible to specify using QIDL. In many cases it is useful to explicitly be able to define a specific level of QoS, for instance for the frame rate of a video flow to be at least 25 frames per second. Being on the type level only, such specifications are not possible to express in QIDL.

Finally, QIDL does not support specification of the semantics of the QoS characteristics. QoS characteristics in QIDL are only defined in CORBA IDL, leaving how these relate to the concepts in the model unspecified.

4.4.6 QuO

In a number of publications [Zinky et al., 1995, Zinky et al., 1997, Loyall et al., 1998, Vanegas et al., 1998], Quality of Service for Objects (QuO) is presented. QuO is an architecture that supports QoS at the CORBA object layer and bridges the conceptual gap between network-level guarantees and application-level requirements. QuO focuses on connections between distributed objects. Such connections may contain behaviour that adapts applications to prevailing conditions. An end-to-end QoS contract consists of two parts: the QoS the client requires from the server and the usage pattern the client promises to abide by. QoS contracts define (using Contract Definition Language - CDL) what QoS characteristics (termed system properties) are observable. Each such system property is a QoS dimension, and QuO divides this multi-dimensional QoS space into regions defined by predicates involving system properties. Two levels of region are defined: negotiation regions and reality regions. Negotiation regions define the system conditions within which the client and server will try to operate. Within a negotiation region there may be many reality regions that are defined in terms of measured system conditions. Transitions between regions may trigger handler routines that can adapt system behaviour. Transitions between reality regions may trigger handlers to take compensating actions to keep the system within the same negotiation region. Transitions between negotiation regions are more severe since compensating actions are not able to keep the system behaviour within the same negotiated region, and client and server objects are informed. QuO reduces variance in system properties by layering. Attempts are made to handle QoS within each layer, but if the system conditions vary too much, transitions between negotiation regions for that layer occur and the layer above is notified. In QuO, design decisions are exposed so that adaptation schemes can chose appropriate implementations when conditions change. The Resource Description Language (RDL) abstracts the physical resources used by the servers while the Structure Description Language (SDL) defines the internal structure of the server and how it consumes resources. This is based on the work on open implementations where reflection is a common technique. Aspect-oriented programming also inspires QuO and future work includes incorporating some of those ideas. From the three languages CDL, RDL and SDL that constitute the Quality Description Language (QDL) suite, code is generated that helps application developers in making their application QoS-aware. The QuO development suite includes compilers and libraries for instrumentation, replication etc.

QuO (with its QDL suite) satisfies a number of the requirements identified in section 4.3. In addition to having a contract-based approach, the specification of negotiation regions with transition behaviour in QuO is useful to support pre-defined adaptation. However, adaptation in QuO is confined to the region transitions specified in CDL; dynamic adaptation is not supported. In particular, service providers have no opportunities to issue new QoS offers that better suit the potential clients since CDL does not include any concept of worth functions.

Reality regions are useful to support maintenance of the promised QoS level, but to specify those in the externally visible part of an object breaks encapsulation. How objects choose to maintain their promised QoS level should not be of any concern to users of the service unless external visibility is needed for negotiation. However, negotiation regions are used for this purpose, reality regions are only used for handling local adaptation where the client does not need to be informed. Hence, reality regions should not be included in the part of the specification intended for clients. Reality regions should instead be a concern of implementers of services when defining the internal behaviour to maintain their promised QoS level.

System condition objects in QuO are objects that measure and control QoS and may be defined locally in a contract or passed as parameters to the contract. In a sense, each system condition object represents a QoS characteristic. However, not only the type of the system condition object is given as a parameter (to indicate what QoS characteristics the contract constrains), but also its implementation class. This is again breaking encapsulation; it should not be of any concern to the client how the server chooses to implement its system condition objects. Both this and the previous issue show that QDL mixes declarative specification of QoS with implementation specification, making QDL unnecessary complex.

QuO is focussed on connections between clients and servers, and supports only numeric and measurable QoS dimensions. This is a constraint that restricts the types of QoS characteristics to be defined, making it unnatural to specify characteristics such as failure semantics (e.g., halt, initial state, rolled back), subjective QoS characteristics such as productivity or satisfaction, or information quality characteristics such as accuracy and validity.

QuO does not support specification of contract types. Contracts in QDL specify constraints on system condition objects, whereas a contract type would only identify what system condition objects to constrain, leaving the specific constraints to each individual contract instance. Not supporting contract types prevents reuse of QoS specifications; new contract instances have to be fully redefined, even if they characterise the same QoS characteristics.

Finally, QuO does not support precise specification of QoS. System condition objects represent the QoS characteristics to measure, but it is not possible to precisely specify what such a characteristic represents in terms of system properties.

4.4.7 QML

In a number of publications [Koistinen, 1997, Frølund and Koistinen, 1998a, Frølund and Koistinen, 1998b, Frølund and Koistinen, 1998c, Koistinen and Seetharaman, 1998], Frølund, Koistinen et al. present their approach to handling QoS in distributed object systems. Their work is focused on specification of QoS. They have designed QML (QoS Modeling Language) that is a general-purpose language for defining QoS properties. QML has three main abstraction mechanisms for QoS specification: contract type, contract and profile. A contract type represents a specific QoS category such as performance or reliability, and contains a number of QoS dimensions. A contract is an instance of a contract type and represents a particular QoS specification for the identified dimensions. Profiles associate contracts with interfaces or elements thereof.

QoS specifications are available both at design time (in QML) and run-time. The QoS fabric (termed QRR - QoS Runtime Representation) enables dynamic specification and manipulation of QoS specifications. Also, it is possible to check conformance between specifications using QRR.

QML satisfies many of the requirements identified in section 4.3. Most notably, it is a general-purpose QoS specification language that separates specification of QoS from specification of functional aspects (in IDL). However, although profiles define QoS offers of service providers, it is not possible to specify a number of profiles that a service provider may offer depending on the prevailing environmental conditions. This static approach to adaptation as supported by QuO is useful since it facilitates the ability to adapt without engaging in potentially costly re-negotiation procedures. Also, QML does not support precise specification of QoS; QoS characteristics (termed dimensions in QML) are only defined as having a value domain, they cannot define how these values relate to the application model.

4.4.8 QDL

In [Daniel et al., 1999], an ODP-based approach to a QoS framework is presented. This approach includes QDL (QoS Definition Language) that can be used to define QoS relations by defining what are called QoS objects. A QoS object contains a QoS expectation and a QoS obligation (QoS offer). A QoS obligation contains a number of properties classified as either simple or complex properties. A simple property is just a name-value pair while a complex property depends on other properties (of other QoS objects). A QoS requirement is specified as a constraint over properties of other QoS objects. QDL uses OCL to specify QoS relations. Properties are defined using OCL types, and constraints in QoS requirements are specified as OCL constraints. QoS objects are instantiated into QoS instances, so QoS objects are essentially types of QoS characteristics, while QoS instances have assigned values to the properties of its type. In [Daniel et al., 1999], composition of QoS objects is also discussed. Two types of compositions are identified, co-operative and concurrent. Co-operative composition is when a QoS object uses another QoS object to provide its QoS offer, while concurrent composition is when one QoS object uses several QoS objects to provide its QoS offer. Dependencies between QoS objects in different combinations of dependency are identified and represented in the QoS framework to be used for QoS management. The approach of QDL is similar to the approach in [Aagedal, 1998] where OCL constraints are also used to specify QoS relations. However, QDL extends this approach by identifying different types of properties and using these to derive dependencies between QoS objects. Also, QDL is independent of any IDL whereas the approach in [Aagedal, 1998] was to integrate with TINA ODL.

From [Daniel et al., 1999], QDL seems immature. For instance, it is not specified how QDL integrates with the application model (e.g., expressed in an IDL). One can, however, easily imagine a similar approach to MAQS QIDL with an extension to IDL (such as the `withQoS` keyword in QIDL). Alternatively, parts of the application model can be included in the specification of QoS objects to link the functional and non-functional specifications, but this requires revision of QDL. Furthermore, adaptation is not addressed in the presentation of QDL. The dependency links that are inserted into the QoS framework can be used to identify whether a change in a property value will influence other QoS objects, but it is not discussed how one specifies adaptation or how the run-time representation of the specification can be used during negotiation. However, QDL has some interesting aspects, in particular the approach to QoS object composition where dependencies are identified and classified for use by the QoS framework.

4.4.9 Evaluation Summary

This subsection summarises the evaluation of existing approaches that has been provided in this section. Each requirement from section 4.3 is listed and it is indicated to what extent each approach supports it. "++" indicates that it is well supported while "--" indicates that it is not supported. "+/-" indicates that some aspects are supported or that some methods support it (in the case of formal methods). ADLs are not included in this summary since these are not targeted towards specification of QoS.

Requirement	Formal methods	SM-IL	TINA ODL	MAQS QIDL	QuO	QML	QDL
1. Generality	+/-	--	+	+	+	++	+
2. Life- cycle support	+/-	--	+	+	+	++	+
3. IDL integration	-	--	+/-	+/-	+	++	?
4. Platform-independence	++	--	++	++	++	++	++
5. RM-ODP compliance	+/-	--	++	+	+	+	+
6. Object-oriented	+/-	+	++	++	+	++	++
7. UML integration	--	--	-	-	-	+	-
8. Separation	+/-	+	--	--	-	++	++
9. Syntactic separation	+/-	--	--	--	++	++	?
10. Predefined adaptation	+/-	+	--	--	++	--	-
11. Negotiation	-	--	--	--	--	+	--
12. Adaptive behaviour	-	--	--	--	++	--	--
13. Composition	+/-	+	--	--	--	--	+
14. Refinement	-	--	--	++	++	++	++
15. Typed	+/-	+	-	++	++	++	++
16. Precision	++	--	--	--	--	--	--
17. Interface annotation	+/-	--	--	++	++	++	?
18. Component annotation	+/-	--	--	--	++	++	?
19. Combinations	+/-	--	--	--	--	--	?
20. Expectations and offers	+/-	--	--	--	+	+	++
21. Code generation	+/-	--	+	++	++	+	+
22. Agreement process	+/-	--	-	-	-	++	?
23. Adaptation	+/-	+	--	-	++	+	-
24. Observable	+/-	++	?	+	++	++	?
25. Comparison	++	--	-	+	+	+	?

Table 2: Evaluation of existing approaches

From the table it can be seen that QML satisfies most requirements of the evaluated approaches. It only lacks support for precise specification of QoS and support for predefined adaptation and adaptive behaviour. It also does not include support for composition of QoS specifications.

4.5 Fundamentals

In this section, fundamental concepts used to define the CQML language are presented. A computational model that is used as a basis for CQML is outlined.

4.5.1 Role of a Computational Model

In order to define a QoS-aware modelling language, one needs a well-defined terminology with an underlying computational model covering the essential parts of what such a language is to support. There is not (and will never be) a final set of terms that covers an area like this exhaustively. Nevertheless, one needs precise and consistent definitions to uniquely specify what the language covers. The computational model presented herein is based on RM-ODP and extensions to it presented in [Blair et al., 1997].

The role of the computational model is to provide an abstract programming model independent of underlying infrastructure implementations. This model represents the set of concepts used for specifying application models, and appropriate mappings between the chosen specification language and the computational model should be provided. Also, the computational model should provide abstractions of the relevant programming concepts from the infrastructures such that these can be used in the application models.

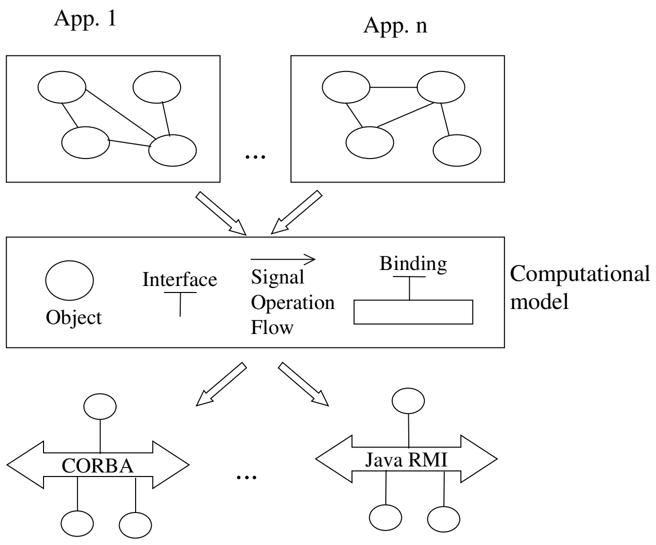


Figure 10. Computational model

Figure 10 illustrates the role of the computational model. Different models of applications use the concepts introduced in the computational model (or their representation in the specification language), and these concepts can be implemented by several infrastructures such as CORBA or Java RMI, or extensions thereof.

4.5.2 Time and Time Models

Time is an important aspect of a distributed system, in particular when considering QoS, and a representation of time is therefore needed.

The nature of time has been an issue of investigation since the ancient Greeks. For our purposes (and in the tradition of computer science), we make some assumptions based on an everyday understanding of the phenomena, leaving philosophical issues aside. In the real world, time is continuous. In computer systems, continuity is approximated due to the fundamental binary properties of such systems. All computer systems have a notion of time, but how it is modelled may differ. There is a major distinction between systems where a simple ordering of events is sufficient and those where quantitative time is needed. Event-ordering is sufficient to prove liveness properties, but in many systems this is of little practical value. That something good eventually will happen is not a satisfactory condition if one has to wait for years for it to happen. Using a simple ordering of events, one cannot specify how long an operation may take; there is no reference to real time. The alternative approach is to extend the ordering of events with a quantitative measure of time. One such measure is real time, that is, time as experienced by humans (wall clock time). QoS, often being time-dependent, requires a quantitative measure of time.

To have a clear mapping from model to realisation, we choose to model time as a discrete domain of time points with a linear ordering. This means that for two unequal points in time (instants) *t* and *t´*, *t* is either before or after *t'*. We also assume that the sequence of time points is equally spaced so that the time periods between any two consecutive points in time are equal (termed chronons). This also means that between two consecutive points in time, there is a time period that cannot be modelled unless we choose a finer granularity of time. Such a sequence of time is isomorphic to non-negative integers, so we can represent the time domain *T* as *T* = {0, 1, …, `now`, …, ∞}. The symbol 0 denotes the relative beginning, and `now` is a special symbol that advances as the clock ticks. An interval is defined as the time between two instants. Duration of time is defined as the length of an interval. A time span is defined as a directed duration of time.

In the real world, we generally only use one time dimension (e.g., co-ordinated universal time - UTC). However, for a system consisting of many components with different timing requirements, the representation of time in one dimension might not be enough. We may need different time domains for different components. [Sabata et al., 1997] presents a taxonomy of time in databases where three different time domains are identified: valid time, transaction time and user-defined time. Valid time represents a domain where a point in time corresponds to when a fact was, is, or is believed to become true in the modelled reality. Valid time can take values from all of *T* so that also future events expected to happen can be modelled. Transaction time represents a domain orthogonal to valid time where a point in time corresponds to when a computer system (originally a database) registers some event. Transaction time can take values from when the computer system was installed until `now` `{0, …, now}`. Note that if the computer system supports both transaction time and valid time, there can be registered future values in transaction time (i.e., it is registered at instant `(now - m)` that at instant `(now + n)`, some fact will become true in the modelled reality, `m,n` $\in \mathbb{Z}^*$). User-defined time is an uninterpreted domain that can be used for any purpose.

In the temporal data model TOOMM (Temporal Object-Oriented Multimedia data Model) [Goebel et al., 1999], a new time dimension termed play time, is introduced as a tailored time dimension for multimedia data types. TOOMM distinguishes between play time-dependent and play time-independent data types. Play time-independent data types include images, text, and graphics, while play time-dependent multimedia data types include audio, video, and animations. Note that play time-independent data types may still have temporal characteristics that enable the modelling of data history and versions. Play time-dependent data types are further classified as streams or computer generated multimedia (CGM) data. Streams are arrays of sequential data elements of a periodic nature, while CGM's are sequences of aperiodic operations that a computer performs over a period of time (e.g., animations). Essentially, the play time dimension represents the time from when a multimedia presentation starts, and all constituent multimedia objects have values in the play time dimension (e.g., start time, stop time, duration, etc.). For a media type like video, this would mean that each frame is timed from the instant the video is started. A chronon in this dimension would be the frame rate, e.g., 1/25s for PAL. For a more complex multimedia object, like a film that consists of both audio and video, the chronon (in TOOMM termed Master Time Unit Duration) of the combined play time dimension is the greatest common divisor of its constituent chronons. CGM objects are aperiodic and use essentially a continuous time domain. To represent such objects, we must chose an appropriate chronon so that the sequence of operations can be mapped onto instants on that time line. If CGM objects are combined with other play time-dependent objects, a common chronon must be chosen.

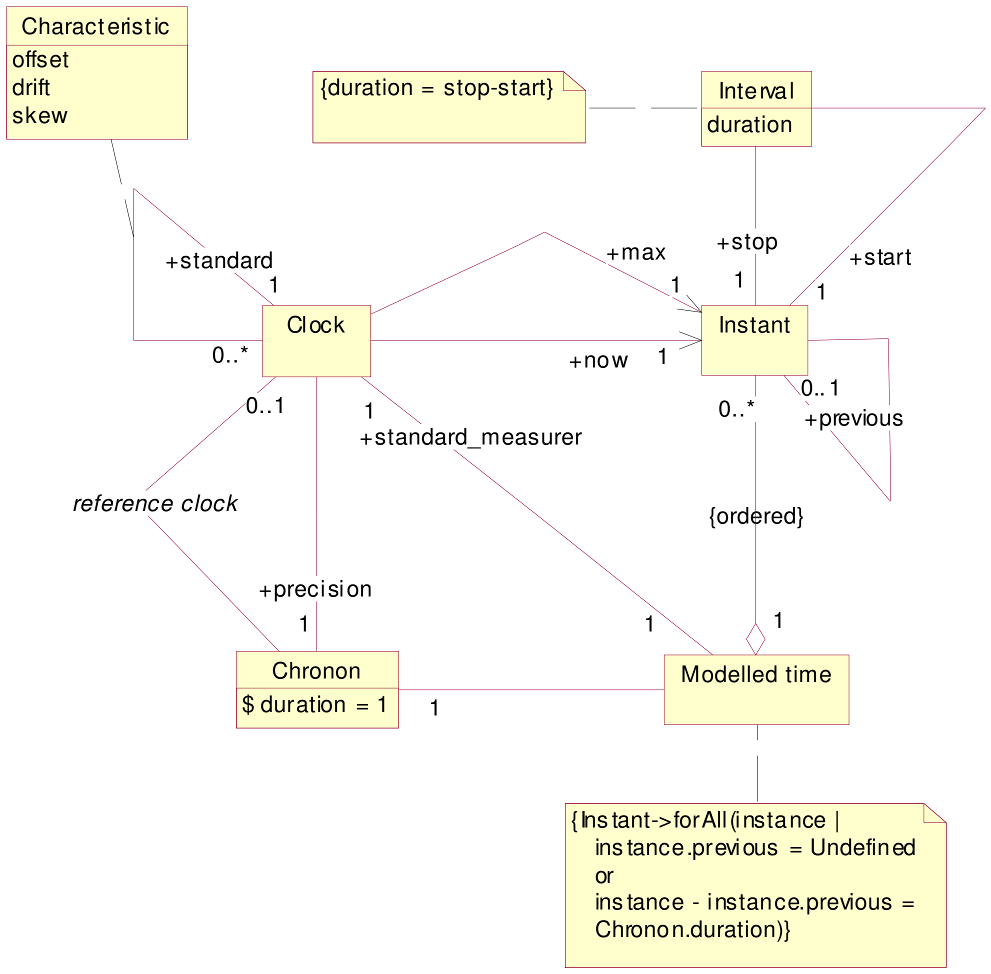


Figure 11: Our model of time

A model of our notion of time is shown in Figure 11. The model shows that the modelled time is an ordered sequence of equally timespaced instants. Each instance of a time dimension (modelled time) is measured by a standard clock. Other clocks may also measure this time dimension, but they are all related to the one standard measurer of time. Since we cannot perfectly synchronise clocks in a distributed system (due to communication delay), the non-standard clocks are related to the chosen standard clock with skew, drift and offset. Since any time domain is isomorphic to the non-negative integers, we define a chronon to have duration of one. This chronon may refer to a reference clock that measures the reference time domain. For instance, a video may have a chronon of 1/25s, i.e., each measurable unit of time in the video time dimension is 1/25 of a second in real time.

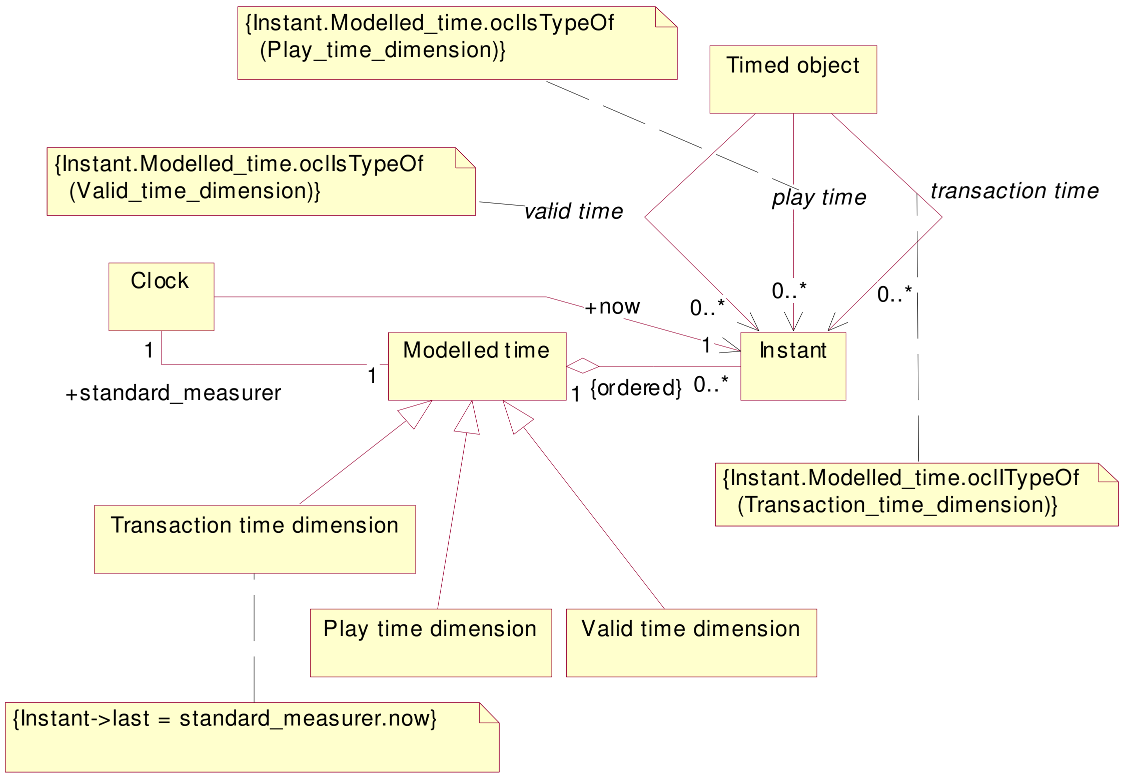

{Instant.Modelled_time.oclIsTypeOf (Play_time_dimension)}
Timed object
{Instant.Modelled_time.oclIsTypeOf (Valid_time_dimension)}
valid time
play time
transaction time
Clock
+now
0..*
0..*
0..*
1
Instant
1
Modelled time
1
+standard_measurer
1
{ordered}
0..*
{Instant.Modelled_time.oclITypeOf (Transaction_time_dimension)}
Transaction time dimension
Play time dimension
Valid time dimension
{Instant->last = standard_measurer.now}

Figure 12: Time dimensions

Figure 12 shows the different time dimensions and how these relate to objects that are timed. A timed object may have associated play time instants, valid time instants and transaction time instants. Such instants may for instance be start and stop time in the play time dimension for a multimedia object, or a number intervals (start and stop instants) in the valid time dimension representing when a fact is believed to be valid.

4.5.3 Objects and Interfaces

The following definitions, based on RM-ODP, are used:

- *Object*: A model of an entity. An object is characterised by its behaviour and, dually, by its state. An object is assumed to be encapsulated; that is, any state changes result from purely internal actions or an observable occurrence at the object's boundary that is explicitly modelled. An object has identity.
- *Behaviour (of an object)*: A collection of actions with a set of constraints on when they may occur.
- *State*: At a given instant, the condition of an object that determines the set of all sequences of actions in which the object can take part.
- *Interface*: An abstraction of the behaviour of an object that consists of a subset of the interactions of that object together with a set of constraints on when they may occur.

Object is the basic modelling concept. Objects interact with their environment (i.e., other objects) through interactions on interfaces (i.e., the message-passing model). Objects have a (possibly dynamically changing) set of interfaces. That an object can have several interfaces represents an important abstraction mechanism since the behaviour of an object can be partitioned into logically distinct sets that other objects can use. Interfaces can consist of incoming or outgoing interactions where the object supporting the interface is either server or client, respectively. An outgoing interface of one object (client) must be matched by an incoming interface for another object (server) for the interaction to take place. The matching of interfaces can be performed by a trading function.

We assume maximum distribution, that is, unless otherwise stated, objects are distributed so that each object may be in a different location from the other objects in its environment.

4.5.3.1 Signals, Operations and Flows

Given the assumption of maximum distribution, interactions between objects by default take time and may fail. We adopt the definition of asynchronous systems in [Hadzilacos and Toueg, 1994] where such systems are defined as systems that make no assumptions about the time intervals involved in any execution (such as process execution speed, message transmission delay or clock drift). In a general-purpose distributed environment, object interaction is asynchronous as there is always a time interval of unspecified length between the occurrence of an event at one location and the occurrence of a causally related event at a different location. The property of asynchronous interactions de-couples in time the client from the server. Note that synchronous systems can be built of components (operating system, networks, clocks) that guarantee timely behaviour. The following definitions are used:

- *Operation*: An interaction between a client object and a server object that is either an interrogation or an announcement.
- *Interrogation*: A two-way operation comprising an invocation, later followed by a termination that carries results back to the originator.
- *Announcement*: A one-way operation comprising an invocation only.
- *Signal*: An atomic shared action resulting in one-way communication from an initiating object to a responding object. Pure signals convey the occurrence of events. Valued signals are signals with associated data values.
- *Event*: An occurrence of interest. Events occur at the boundary of an object and at a particular instant.
- *Flow*: An abstraction of a sequence of interactions, resulting in the conveyance of information from a producer object to a consumer object.

Operational interfaces consist of operations, signal interfaces consist of signals and stream interfaces consist of flows. Objects may have any number of each of these kinds of interfaces. Also, objects may have heterogeneous interfaces containing a mix of operations, signals or flows.

An operation is represented in two interfaces: the outgoing interface of the client object and the incoming interface of the server object.

Each invocation or termination results in an event. Hence, an interrogation results in four events:

1. the invocation emitted from the client,
2. the invocation received by the server,
3. the termination (reply) emitted from the server, and finally
4. the termination received by the client.

A flow results in sequences of events at the producer and consumer interfaces corresponding to each information element conveyed.

4.5.3.2 State and Behaviour

The state of an object is the abstract data it contains. The current state represents the accumulated condition of that object from the initial state through all actions it has participated in. In each state, an object offers one set of interfaces. States are discrete and the transition from one state to the next is instantaneous. The following definition is used:

- *Trace*: A record of an object's interactions, from its initial state to some other state.

At any point in time, the state of an object determines the set of behaviours in which the object can take part. In other words, the definition of states and their possible transitions represents the protocol of the object, and a trace represents one possible sequence of interactions that leads to an allowable sequence of states according to its protocol. The history of an object o_i is characterised as a sequence of events corresponding to its interactions:

$$history(o_i) = h_i = e_i^1 e_i^2 e_i^3 \ldots$$

Note that the series of events corresponds to actions resulting from the interactions ending in a state, it is therefore a one-to-one correspondence between a series of events and a series of states:

$$e_i^1 e_i^2 \ldots e_i^k \leftrightarrow s_i^1 s_i^2 \ldots s_i^k$$

From the point of view of a single object, its history is a sequence of events ordered by its local clock. If we define a timing function $\tau(e)$ of an event, the following holds for history h_i:

$$\tau(e_i^1) < \tau(e_i^2) < \tau(e_i^3)\ldots$$

We are sometimes interested in only an initial prefix of the history:

$$h_i^k = e_i^1 e_i^2 e_i^3 \ldots e_i^k$$

Note that if an object has no internal actions, the trace of an object to state s_i^k is h_i^k.

The trace of an object can be partitioned into distinct subsequences of the trace corresponding to each operation, signal or flow in the interfaces of the object. We project the history of the interactions through interface i_j of an object o_i as:

$$h_i / i_j = e_{ij}^{n_1} e_{ij}^{n_2} e_{ij}^{n_3} \ldots$$

where an event $e_{ij}^{n_m}$ is the n$_m$'th event of object o_i and the m'th interaction through interface i_j of the object. Hence, $\tau(e_{ij}^{n_m}) < \tau(e_{ij}^{n_{m+1}})$ for all m, or, in other words, the ordering is preserved in the projection. Similarly, we project the history of the interactions of operation op_k in interface i_j of an object o_i as:

$$h_i / i_j .op_k = e_{ijk}^{n_1} e_{ijk}^{n_2} e_{ijk}^{n_3} \ldots$$

The four events resulting from an interrogation are represented in the trace as follows. The invocation event emitted from the client is recorded in a sequence of events of service emittances. This is represented in an outgoing interface of the client object and termed `SE` (service emittances). The reception of the invocation by the server object is recorded in a sequence of events of service receptions. This is represented in an incoming interface of the server and is termed `SR` (service receptions). The termination event emitted from the server object is recorded in a sequence of events of service emittances. This is represented in the same incoming interface of the server object and termed `SE`. The reception of the termination by the client object is recorded in a sequence of events of service receptions. This is represented in the same outgoing interface of the client object and is termed `SR`.

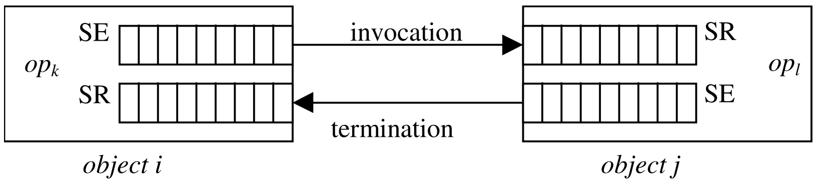


Figure 13: Interrogation events

In Figure 13, an interrogation from the invoking operation op_k in the client object *object i* to the performing operation op_l in the server object *object j* is shown. The sequences of SE and SR events are also shown for both the invoking and the performing operation. If *se* denotes the emittance of an invocation, SE for operation op_k in interface i_j of an object o_i can be defined as:

$$SE_{ijk} = h_i / i_j .op_k .se = e_{ijk}^{n_1} e_{ijk}^{n_2} e_{ijk}^{n_3} \ldots$$

where $e_{ijk}^{n_m}$ is the n_m'th event of object o_i and the m'th invocation of operation op_k through interface i_j of the object. Similarly, if *sr* denotes the reception of an invocation, SR for operation op_k in interface i_j of an object o_i can be defined as:

$$SR_{ijk} = h_i / i_j .op_k .sr = e_{ijk}^{n_1} e_{ijk}^{n_2} e_{ijk}^{n_3} \ldots$$

Note that SE and SR are defined for signals and flows also. Note also that an invocation always has an event in the SE sequence of the client with a corresponding event in the SR of the server. Analogously, a termination has events in the SE and SR of the server and client, respectively.

The following properties hold:

$$h_i / i_j .in_k = h_i / i_j .in_k .se \oplus h_i / i_j .in_k .sr$$

$$h_i / i_j = \bigoplus_{k=1}^{n} h_i / i_j .in_k \text{ , } n = \#\, interactions$$

$$h_i = \bigoplus_{j=1}^{n} h_i / i_j \text{ , } n = \#\, interfaces$$

where ⊕ denotes a sequence merge operation in which the ordering of the events in the sequences are preserved according to the ordering of the object history, and in_k is either an operation, a signal or a flow in interface i_j.

For each operation, signal or flow in an interface of an object, SR and SE represent its use in the lifetime of the object. The merger of all SE and SR sequences in all interfaces of an object yields the trace of that object.

4.5.4 Types and Instances

A type is a predicate characterising a collection of objects, i.e., it defines common properties of a set of objects (the extent). If the predicate holds for an object, the object is of that type (i.e., the object is a member of the extent). Since an object can satisfy many predicates, an object can conform to many types. The predicate may refer to both state and behaviour (e.g., state changes), and may put restrictions on both the object and its environment. As the computational model defines, a system consists of interacting objects. Such objects are instances. At any point in time, an instance has a state where each value type is assigned a specific value. While a type defines the value space any instance of that type must have, the instances have individual values.

4.5.5 Bindings

We assume maximum distribution and asynchronous communication between objects. However, bindings are introduced to model communication channels that bridge object locations. Binding objects represent explicit bindings created upon request from an application object (as opposed to implicit bindings that are created by the infrastructure). The client and server objects that communicate through a binding object, communicate asynchronously with each other, but they may communicate synchronously with the binding object as it is represented on both locations. In this case, the binding represents the communication delay and communication failure semantics. Binding objects can represent different kinds of communication semantics such as at-least-once or at-most-once.

Binding objects have at least two interfaces; one incoming interface the client can use and one outgoing interface through which the server is invoked. These interfaces correspond to the incoming interface of the server and the outgoing interface of the client, respectively. Hence, the client can use the binding object as if using the server directly and the server is invoked by the binding as if the client invoked it directly; the binding object only acts as an intermediary.

4.5.6 Synchronous and Asynchronous Objects

Actions of objects take an indefinite amount of time. However, in order to manage timing constraints, we must ensure deterministic behaviour. We therefore distinguish between synchronous and asynchronous objects. Synchronous objects respond instantaneously to requests according to the synchrony hypothesis [Berry and Gonthier, 1988] and are sometimes called reactive objects. We can use reactive objects to anchor the timing dependencies down to deterministic behaviour, and thereby use these as managers of timing constraints without themselves having to be managed. All other objects are asynchronous and use an unpredictable amount of time to perform any action.

4.5.7 Formalisation

Essentially, the computational model presented herein is a message-passing model similar to the basic computational model from RM-ODP with the extensions developed at Lancaster University and CNET, France Telecom as presented and discussed in [Stefani, 1993, Blair et al., 1997, Blair and Stefani, 1997, Najm and Stefani, 1997, Leboucher and Najm, 1997]. The model of Blair, Stefani et al. has been formalised using transition rules (rewrite rules) to specify allowable state changes, using an axiomatic approach, and using the temporal logic of actions (TLA) in the references cited.

4.6 CQML

This section presents CQML. We present each specification construct in CQML by discussing its properties, defining its syntax, by showing its use in examples, and finally by defining a model of the concepts in the language (meta-model).

4.6.1 Overview

CQML includes four types of specification constructs. The basic construct is *QoS characteristic*. QoS characteristics are defined as user-defined types. QoS characteristics are then grouped and restricted into specifications of QoS. For this, *QoS statements* that constrain each constituent QoS characteristic within specific ranges of values are defined. These two constructs focus on specification of QoS independent of what interface it annotates and how the QoS mechanism is implemented. A third construct, *QoS profiles*, relate QoS statements to specific components or parts thereof. Finally, *QoS categories* are used to group any of the three aforementioned concepts.

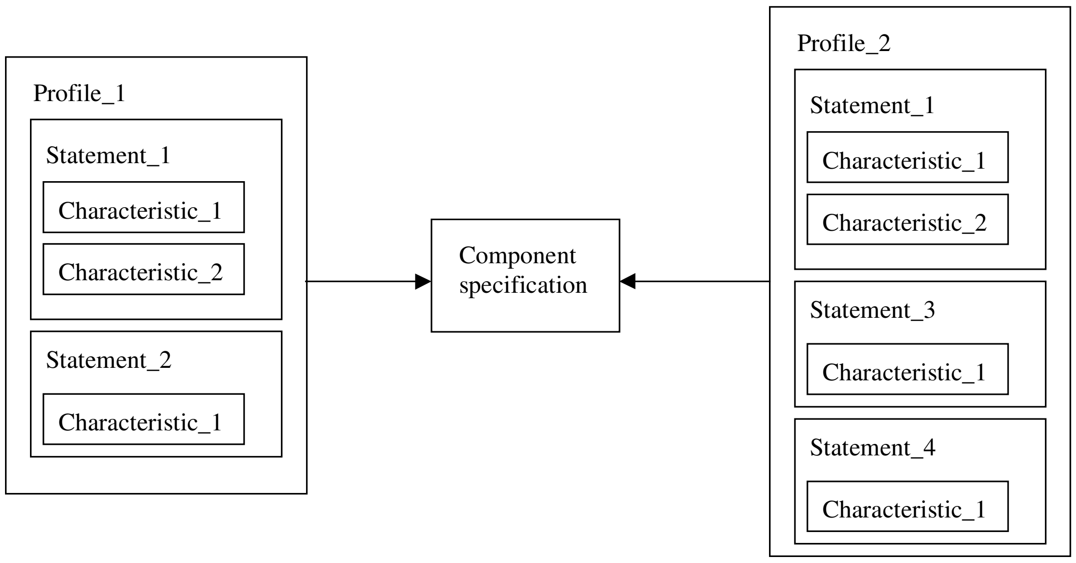


Figure 14: CQML overview

Figure 14 illustrates that a component specification may have several QoS profiles assigned to it, and that each such profile uses a number of QoS statements. A QoS statement constrains any number of QoS characteristics.

4.6.2 Notation

CQML keywords in examples, grammar and text are in **bold**. Code in examples and when referenced in the text is in the `Courier` font. The grammar is presented using EBNF syntax where angle-brackets enclose non-terminals, '[' and ']' enclose an optional part, '|' means choice, '*' represents zero or more times, '+' means one or more times, and '::=' denotes productions. Curly brackets ('{' and '}') are used for grouping while a pair of single quotes ('') are used to enclose terminals. Non-terminals in the OCL grammar are referred to using 'OCL::' as prefix.

4.6.3 QoS Characteristics

A QoS characteristic is the basic building block of a QoS specification, and we present this first.

A QoS characteristic has a unique name that is used for type matching.

A QoS characteristic has a domain in which it is quantified. A QoS characteristic defines a type based on one of the fundamental types **`numeric`**, **`set`** or **`enum`**. Each such fundamental type defines a domain (set of possible values) that constrains an instance of that type.

4.6.3.1 Basic QoS Characteristics

Declarations of basic QoS characteristics follow this pattern[3,4]:

<characteristic_declaration>	::=	**quality_characteristic** <characteristic_name> '{'<characteristic_body> '}'
<characteristic_body>	::=	<domain_declaration>
<domain_declaration>	::=	**domain** ':' <domain_type> ';'
<domain_type>	::=	**numeric** | **set** | **enum**

As an example, a basic QoS characteristic can be declared as:

```
quality_characteristic output {
  domain: numeric;
}
```

We have here defined a basic QoS characteristic named `output` that has values in the numeric domain. The fundamental type **numeric** is the set of all real numbers. A QoS characteristic based on the numeric domain may restrict it to contain only the set of integers or natural numbers. By default, CQML restricts values of the numeric domain to be from the set of natural numbers (including zero). For values to be in the other (more general) domains of all integers or real numbers, it must be explicitly declared. For example, to define a QoS characteristic `output_frequency` where the values are real numbers, the default numeric domain must be extended to be that of real numbers:

```
quality_characteristic output_frequency {
  domain: numeric real;
}
```

A QoS characteristic based on the numeric domain may also be restricted by specifying upper and/or lower bounds. For the countable domains natural numbers and integers, a boundary restriction restricts the number of values an instance can have, while for the uncountable domain of real numbers, it only restricts the value range of the instances. Boundary restrictions are either "from/until and including" or "from/until, but not including" a limit (using '[' or ']', or '(' or ')', respectively). In a computer system, real numbers are always approximated onto a certain level of precision given by the number of bits that represent them. In CQML, we do not restrict the level of precision of real numbers or the range of natural numbers and integers. However, since such restrictions exist in any computer system, the type system of CQML must be mapped onto the target type system of the computer system when specifications in CQML are used to develop applications on the target platform.

The grammar for the numeric domain type is then:

<domain_type>	::=	<numeric> | **set** | **enum**
<numeric>	::=	**numeric** [<type_specification>] [<boundary_restriction>]
<type_specification>	::=	**real**
	|	**integer**
	|	**natural**
<boundary_restriction>	::=	<lower_boundary> <OCL::number> '..' [<OCL::number>] <upper_boundary>
	|	<lower_boundary> '..' <OCL::number> <upper_boundary>

[3] Note that we use the term **quality_characteristic**, not `qos_characteristic`. We choose this to avoid confusion when defining characteristics of quality in use since the `qos_` prefix may lead modellers to think of the service irrespective of its use, which is not the intention of quality in use characteristics.

[4] Only simplified parts of the grammar are presented in this chapter. The complete grammar is provided in appendix III.

```
<lower_boundary>        ::=  '[' | '('
<upper_boundary>        ::=  ']' | ')'
```

To restrict the output frequency to only be positive numbers or zero, it can be redefined as:

```
quality_characteristic output_frequency {
  domain: numeric real [0..);
}
```

Note that this QoS characteristic does not have an upper limit, an arbitrarily large frequency can be modelled.

If a domain is ordered, one may define a "higher-quality-than" relation between elements such that one can identify what value represents the higher quality. If high values of the domain are considered better than low values, the domain is said to be increasing and '>' represents the "higher-quality-than" relation. On the other hand, if low values are considered better, the domain is said to be decreasing and '<' represents the "higher-quality-than" relation.

The numeric domain has the common linear ordering, and the comparison operators '<' and '>' are well-defined in this ordering. The semantics of the ordering can be specified as either **increasing** or **decreasing**, depending on whether the semantic value (worth) increases or decreases when the number value increases (i.e., whether higher or lower values are considered "better"). For integers, also the operators '=', '<>', '>=' and '<=' have a common predefined meaning. However, using the argument above on computer representation of real numbers, these operators can also be used for real numbers on specific target platforms. The numeric domain may be used to specify any scale type (nominal, ordinal, interval, ratio and absolute). The grammar for the numeric domain is extended to:

```
<numeric>              ::=  [<direction>] numeric [<type_specification>]
                            [<boundary_restriction>]
<direction>            ::=  increasing | decreasing
```

Values of the `output` characteristic are from the numeric domain and have thereby a predefined ordering, but the semantics of the difference between high and low values is not defined. If we want to specify the direction of the ordering, we can specify that the domain is increasing, meaning that high values are better than low. We redefine `output` to:

```
quality_characteristic output {
  domain: increasing numeric;
}
```

Instances of the fundamental types **enum** and **set** have values from a set of user-defined names[5]. The difference between the two is that an instance of the **enum** type is one member of the set of user-defined names, while an instance of the **set** type is one member of the power set of the set of the user-defined names.

The **enum** type is defined as a set of user-defined names. Normally, enumerations define a nominal scale type. However, we can add a user-defined ordering (partial or total) of the elements. If the ordering is total, we have an ordinal scale type. If the elements are ordered, we must also specify the direction of the semantic value, i.e., whether the semantic value (worth) increases or decreases when the element value increases. By default, the elements in an enumeration are ordered in the listed order if the worth direction is specified and no explicit order is defined. The ordering of an enumeration is transitive and is specified as a list of ordered pairs of the user-defined names.

[5] The definitions of the fundamental types **enum** and **set** are similar to the definitions in QML by Frølund, Koistinen, et al.

The **set** type is also defined as a set of user-defined names, but an instance may be any subset of this set, not just one element. Similar to enumerations, we may add an ordering (partial or total) of the user-defined names and must then specify the direction of the semantic value. As for enumerations, the user-defined names are ordered in the listed order if the worth direction is specified and no explicit ordering is defined. The ordering of instances of **set** depends on whether the user-defined names are ordered or not. If they are not ordered, the ordering of two instances of **set** (elements of the power set) corresponds to set inclusion. If the user-defined names have an ordering, a set of user-defined names `A` is less than a set of user-defined names `B` if the user-defined names in `A` that are not in `B`, are ordered below at least one of the members in `B` that are not in `A`. This ordering semantics is similar to the ordering semantics in QML. The grammar for the set and enumeration domain types is:

```
<domain_type>          ::=  <numeric> | <set> | <enumeration>
<set>                  ::=  <simple_set>
                       |    <direction> <simple_set> [with <order_definition>]
<simple_set>           ::=  set '{' <name_list> '}'
<order_definition>     ::=  order '{' <one_order> {',' <one_order>}* '}'
<one_order>            ::=  <element_name> <order_operator> <element_name>
<order_operator>       ::=  '<' | '>'

<enumeration>          ::=  <simple_enumeration>
                       |    <direction> <simple_enumeration> [with <order_definition>]
<simple_enumeration>   ::=  enum '{' <name_list> '}'
```

Note that CQML does not include a general-purpose type system in the definition of QoS characteristics. We have excluded user-defined types and enumerations and sets of user-defined types as QoS characteristic domains. To include these would give CQML more expressive power, but would not fit the purpose of a declarative language for specification of QoS. User-defined types may include attributes and operations and are used to model systems. A QoS characteristic, on the other hand, should characterise some aspect of a service that can be identified and quantified. Since only identification and quantification are of interest, the expressive power of user-defined types is not needed and **numeric**, **enum** and **set** suffice.

Optionally, the values of a domain can also have a unit. There are no predefined units. Unit specifications are treated only as informative. However, units are represented in the grammar as non-terminals and a compiler and run-time system for conformance checking can be extended to reason about units (e.g., unit conversions).

For example, `delay` may be a useful QoS characteristic:

```
quality_characteristic delay {
  domain: decreasing numeric milliseconds;
}
```

Here we defined low values of `delay` to be better than high values. We have also specified that delay has the unit of milliseconds, but this has no additional formal semantics.

4.6.3.2 Specialised QoS Characteristics

QoS characteristics can be specified based on other QoS characteristics by adding details. We use specialisation as known from the object-oriented paradigm to achieve this. The substitutability principle underlies specialisation; a specialised QoS characteristic should be possible to use wherever the QoS characteristic it specialises can be used without anyone expecting the base QoS characteristic noticing any difference. To be a specialisation means that the derived QoS characteristic is a subtype of its base type, i.e., all members of its extent are also members of the extent of the base QoS characteristic. Since a specialised QoS characteristic must be a restriction of its base, only similar or more restrictive numeric domains may be used in the definition of a specialised QoS characteristic. One cannot, for instance, specify a QoS characteristic of real numbers to be derived from a QoS characteristic of natural numbers or integers.

We extend the syntax for the declaration of QoS characteristics:

```
<characteristic_declaration>   ::=  quality_characteristic <characteristic_name>
                                    [<specialisation>] '{'<characteristic_body> '}'
<specialisation>               ::=  ':' <parent_name>
```

For instance, the `output` characteristic may be specialised to `frameOutput` as:

```
quality_characteristic frameOutput : output {
  domain : increasing numeric frames/second;
}
```

Here we have only specified `frameOutput` as a specialisation of `output` by adding the unit it is measured in, but without any additional semantics. Since it is a specialisation, we cannot use a different domain unless its values represent a subset of the values of the original domain. Actually, the definition of the domain as **`increasing numeric`** is redundant because it is already defined for `output`, but it must be repeated in order to add the unit specification.

The concept of specialisation will be more useful when we introduce what the values of the domain of the QoS characteristics represent in subsubsection 4.6.3.4.

4.6.3.3 Derived QoS Characteristics

Derived QoS characteristics are defined as functions of other QoS characteristics.

Statistical characteristics are defined by adding statistical aspects. These include maximum, minimum, range, mean, variance, standard deviation, n-percentile, and statistical moments as defined in [ISO/IEC JTC1/SC21, 1995b]. In addition, frequency X is defined as the frequency (in percent) of which the characteristic has value X. Such statistical derivations are defined by adding these aspects to already existing QoS characteristics, or by including them in the definition of new ones. Statistical characteristics are measured over some period of time to decide their value. By default, we assume a Gaussian distribution of the measurements. One may, however, specify that the measurements follow other distributions. CQML supports specification of any distribution by specification of its name with any number of parameters it needs. However, since such distributions are not identified in the grammar, it must be a common understanding between the modeller and the QoS framework that measures the QoS characteristics for these distribution names to have meaning. The distribution name and its parameters only represent hooks for the underlying QoS framework to be able to compute the appropriate statistics. Besides the default Gaussian distribution, the most important distribution for QoS measurements is the Poisson distribution. It is often used to model rare, isolated events which occur at random in time or space, for instance packet arrivals that can be used to compute delay. For this reason, we call this out as a keyword in CQML. Other important distributions such as Weibull, Pareto, etc. are not called out; if such a distribution is to be used it needs to be specified by a name together with any parameters characterising it.

To declare statistical characteristics, we extend the body of a characteristic declaration according to this pattern:

```
<characteristic_body>         ::=  <characteristic_definition>
                              |    <characteristic_derivation>
<characteristic_derivation>   ::=  [<characteristic_definition>] <statistical_derivation>
<statistical_derivation>      ::=  {<one_aspect> ';'}+
<one_aspect>                 ::=  maximum
                              |    minimum
                              |    range
                              |    mean
                              |    variance
                              |    standard_deviation
                              |    percentile <number>
                              |    moment <number>
                              |    frequency {<element_name> | <number>
                                              | <range_values>}
                              |    distribution <distribution>
<range_values>               ::=  <lower_boundary> <elements> <upper_boundary>
<elements>                   ::=  <element> '..' <element >
<element>                    ::=  <OCL::name> | <OCL::number>
<distribution>               ::=  Poisson <mean_number>
                              |    <distribution_name>
                                   [<distribution_parameter> {',' <distribution_parameter>}* ]
<distribution_parameter>     ::=  <element>
```

Using the grammar above, an example is:

```
quality_characteristic statisticalDelay : delay  {
  maximum;
  minimum;
  range;
  mean;
  variance;
  standard_deviation;
  percentile 90;
  moment 3;
  frequency 10;
}
```

The definition above refines the `delay` by adding a number of statistical aspects (of which some are redundant but are, nevertheless, included for illustration purposes) to make a new QoS characteristic `statisticalDelay`. This QoS characteristic includes not only that the delay is measured in milliseconds and that low numbers are preferred (as defined in `delay`), but it adds a number of statistical aspects:

- **`maximum`** represents the highest possible value of the characteristic.
- **`minimum`** represents the lowest possible value of the characteristic.
- **`range`** represents the boundaries between which the values of the characteristic are. This follows directly if both maximum and minimum are defined.
- **`mean`** represents the arithmetic mean of the values of the characteristic.
- **`variance`** represents the statistical variance of the values of the characteristic.
- **`standard_deviation`** represents the statistical standard deviation of the values of the characteristic (follows from the variance).
- **`percentile`** `90` represents the 90th percentile of the values of the characteristic.
- **`moment`** `3` represents the 3rd central moment of the values of the characteristic (rarely used aspect).

- **`frequency`** `10` represents the frequency of a value of exactly 10 of this characteristic.

A Poisson distribution can be specified as **`distribution Poisson`** `10`, where it is specified that the measurements follow a Poisson-distribution with a mean of 10.

Alternatively, the `statisticalDelay` can be defined without direct reference to the `delay` characteristic as (here including only one statistical aspect):

```
quality_characteristic statisticalDelay {
  domain: decreasing numeric milliseconds;
  mean;
}
```

Frequency aspects can be defined for individual values and for ranges of values. Individual values are shown above, while ranges are specified as "**`frequency`** `[10..13)`" meaning that the frequency of values from 10 (including 10) up to 13 (excluding 13).

Statistical derivations specify properties of a collection of evaluations (measurements) of an instance of a QoS characteristic (typically over some period of time). In the definition of a QoS characteristic, the relevant statistical aspects (from the predefined set in CQML) are identified. Depending on the scale type, different statistical aspects can be specified for a QoS characteristic. Only frequencies can be specified for a nominal scale type. For an ordinal scale type, maximum, minimum, range and percentile may also be specified. For interval, ratio and absolute scale types, all statistical aspects can be specified. This means that it is only for QoS characteristics in the numeric domain that all statistical aspects may be specified, depending on which scale type is used. Since unordered enumerations and sets define nominal scales, only frequency is applicable (specifying how often a value must occur). Similarly, since linearly ordered enumerations and sets define an ordinal scale, maximum, minimum, range, percentile and frequency are applicable. The scale type of a numeric QoS characteristic cannot be specified in CQML, it is the modellers responsibility to apply only the appropriate statistical aspects if needed. CQML treats any statistical aspect in the numeric domain as appropriate.

The evaluations (measurements) of an instance of a QoS characteristic are assumed to follow a Gaussian distribution and all statistical aspects are related to this, but others can be specified. Except for the Poisson distribution that is called out as an important distribution, a new distribution is specified as an agreement between the underlying measurement mechanism and the modeller. For instance, a measurement mechanism may support statistical constraints on measurements following an exponential distribution and has in its documentation stated that QoS characteristics following such a distribution can be specified using the production:

```
<distribution>                ::=  exponential <mean_number>
```

Such an extension would be specific to the measurement mechanism since it would need to support this new distribution name and interpret the distribution specification. CQML treats this distribution name only as a string and any parameters as a comma separated list of strings, and it can be made available to the measurement mechanism.

Derived QoS characteristics can also be defined as functions of other QoS characteristics. To include this, we extend the syntax to:

```
<characteristic_derivation> ::=  <function_derivation>
                            |    [<characteristic_definition>] <statistical_derivation>
<function_derivation>       ::=  {<characteristic_name> ';'}+ [<domain_declaration>]
```

Note that the grammar needs to distinguish between specialised and individual QoS characteristics since a statistical derivation needs a domain definition in order to be well-defined (this is not shown in the grammar here, but the grammar in appendix III includes this). Given a definition of `reliability` and `maintainability` characteristics, `availability` can be defined as:

```
quality_characteristic availability {
  maintainability;
  reliability;
}
```

Without defining the semantics of `availability`, the definition above shows only that it depends on `reliability` and `maintainability`, but its domain and how it depends are left unspecified. However, this provides a hook for the underlying QoS framework to define a function that defines the dependencies between the QoS characteristics. Alternatively, such dependencies can be specified in the values-clause as presented next.

4.6.3.4 Invariants and QoS Characteristic Values

Until now we have only defined QoS characteristics as types and specified to what domain their values belong. However, it is also useful to define what the values of the characteristic represent. For this purpose we use the Object Constraint Language (OCL) as defined in [UML RTF, 1999] and presented in [Warmer and Kleppe, 1999].

In the definition of UML, OCL is used to increase precision by specifying constraints on the UML meta-model. This is achieved by specifying pre- and post-conditions of operations and constraints on the modelling elements and their relationships. In CQML, OCL can be used to define values of the QoS characteristics by constraining the values of the domain and by specifying how different aspects relate. OCL is a typed language and has a number of predefined operations on its basic types that can be used. In addition, user defined types can be referenced.

Using OCL, an invariant (a predicate that must always be true) can be specified. In OCL, the keyword `self` is used to refer to the current instance (i.e., `this` in C++ and Java). Using `self` to refer to the instance of the QoS characteristic, the invariant can further restrict the value of the characteristic. For instance, the invariant `self >= 10` would restrict the use of this QoS characteristic in a QoS statement to only be valid above the explicit lower boundary of 10. An invariant is used to specify inherent properties of the QoS characteristic, valid for any measurement. It may for instance be the case for a delay characteristic that any measured delay is at least 10 milliseconds. If a measurement shows less than 10 milliseconds, the measurement is considered erroneous and converted to 10. The invariant above can be used to specify this situation. In most cases, however, the definition of a QoS characteristic only defines the type of aspect that can be identified and quantified, individual constraints of this quantification is done in QoS statements as discussed in subsection 4.6.4.

Invariants are included in the grammar as:

```
<characteristic_definition>::=  <domain_declaration> [<invariant>]
<invariant>                ::=  invariant <OCL::oclExpression> ';'
```

The implicit rule of range being between the maximum and the minimum value when defining statistically derived characteristics can be formalised using OCL:

```
quality_characteristic statisticalDelay : delay  {
  maximum;
  minimum;
  range;
  invariant: range[0] = minimum and range[1] = maximum;
}
```

We have here introduced the keyword invariant after which an OCL expression defines that lower boundary of range (referenced as `range[0]`) is minimum and the upper boundary (referenced as `range[1]`) is maximum. The **`invariant`**-keyword introduces a Boolean OCL-expression that always is to be evaluated to true. This can be used to define properties of this characteristic.

To be able to specify the values of a QoS characteristic in more detail than only to specify their domain, we include the possibility to constrain the values by an OCL-expression in a **`values`** clause. In this clause, the precise semantics of the instances of the QoS characteristic can be defined. Here it is specified what the QoS characteristic represents in terms of properties of the parts of the system it characterises. To be able to define the semantics, the parts must be made available as parameters. A QoS characteristic may therefore have a number of formal parameters in its definition. The formal parameters can be pre-defined or user-defined types. The **`values`** clause is specified using OCL. The OCL expression specifies how the properties of the formal parameters can be combined and compared to provide the value of the QoS characteristic. The result of the OCL expression must have the same type as the QoS characteristic. In a running system, actual parameters can be given to a monitor of this QoS characteristic for specific parts of the system, and values of the QoS characteristic can be evaluated based on the OCL expression. The grammar is:

<characteristic_declaration>	::=	**`quality_characteristic`** <characteristic_name> [‘(’ <OCL::formalParameterList> ‘)’] [<specialisation>] ‘{’<characteristic_body>‘}’
<specialisation>	::=	‘`:`’ <parent_name> [‘(’ <OCL::actualParameterList> ‘)’]
<characteristic_body>	::=	<characteristic_definition>
	|	<characteristic_derivation>
<characteristic_definition>	::=	<domain_declaration> [<values_declaration>][<invariant>]
<values_declaration>	::=	**`values`** ‘`:`’ <OCL::oclExpression> ‘`;`’

However, to be able to specify more complex semantics using OCL, we need a type model to use in such specifications. Until now, when defining a QoS characteristic, we have only had the characteristic under specification available and this limits what can be expressed. To rectify this we take two approaches: to define a set of relevant background types (in addition to the predefined ones from OCL), and to include parameters in the definition of QoS characteristics.

OCL has a predefined parameterisable type, `Sequence`, with the usual attributes and operations for sequences available. Here we introduce `EventSequence` that is a parameterisation of `Sequence` containing elements of type `Event`.

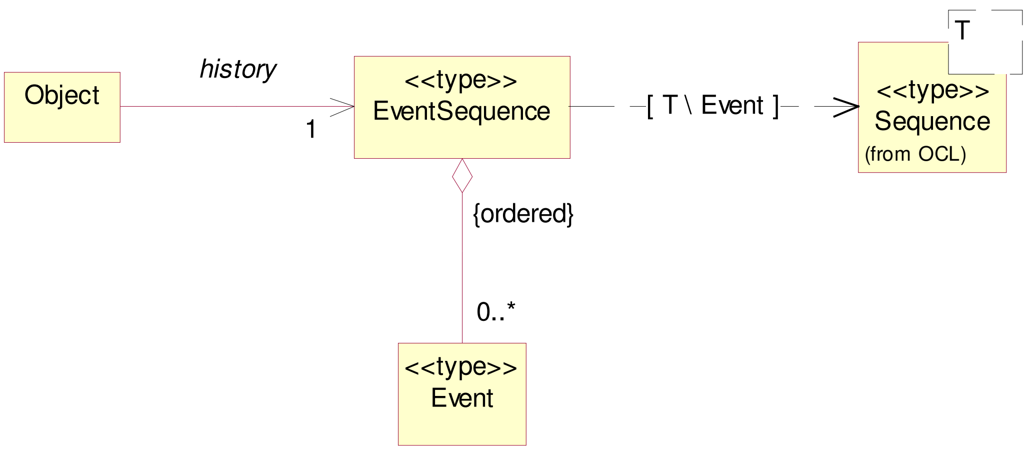


Figure 15: EventSequence

Figure 15 shows a model of `EventSequence` and its relations to other concepts. An `EventSequence` instantiates the parameterised `Sequence` type from OCL using the type `Event` for the generic type `T`. Following the computational model defined in section 4.5, an object has an event sequence that is the history of the object. Such a model provides what is needed if the events contain information that can be used to derive to what part of the object the event pertains (i.e., which interaction in which interface). However, to be more useful in CQML, we explicitly represent these sequences in the event model.

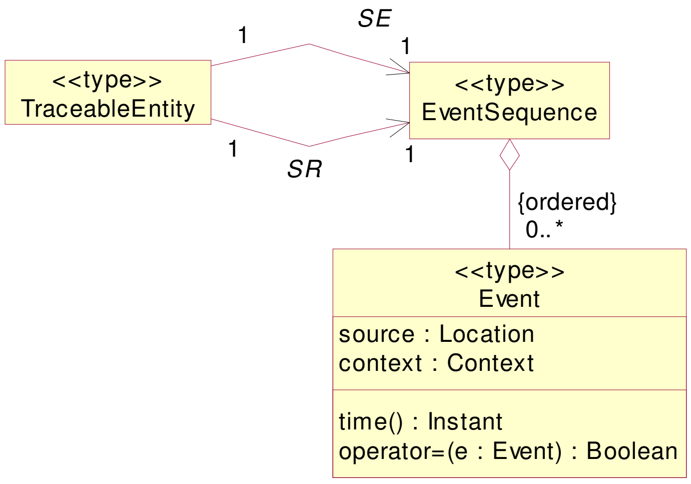


Figure 16: Event model

In Figure 16, we introduce a traceable entity that has two sequences of event termed SE and SR. From the computational model, objects have a history that can be projected onto its interfaces or signals, operations and flows to derive their `SE`- and `SR`-sequences of events, so all these can be traceable entities.

Events have an attribute termed `source` that contains information about the spatio-temporal location of the event source. We assume that Events have an associated time dimension, so the function `time()` can be defined to derive, from `source`, the instant in that dimension when the event occurred. In addition to timing information, `source` also contains the identities of the object, interface and interaction where the event occurred. This is required for instance to be able to decide whether the timing of two events can be meaningfully compared (i.e., that they are measured by the same clock) or to be able to extract sequences of events pertaining to particular parts of the object.

`context` is another attribute of `Event`. It contains information about the event context and is used to identify (parts of) the state of the initiating object. From the computational model, we have that all interactions (including signals) may carry information. In addition to regular parameters, context information of the initiating object may be conveyed. Context information identifies what the values of relevant parts are (e.g., the identification of the frame that is sent or the flow that the frame is a part of). The context attribute can contain any information available at the initiating object that is useful when reasoning about events. In a chain of causally related events, the event context may be cumulative, facilitating reasoning on composites.

We have also introduced an equality operator, this is used to check whether two events are the same events in the same event context. Its use will become apparent in subsubsection 4.6.5.2.

`EventSequence` can be used in a precise definition of a QoS characteristic `startUpTime`. This characteristic has values in the numeric domain representing milliseconds. To be fully defined, it needs a parameter representing the flow that is initiated. This parameter represents an object of a type defined elsewhere, and it can be referenced in the OCL-expressions that define the values of the characteristic. In this case we use an object of type `Flow`, that is, an object that implements the `Flow` interface. `Flow` consists of a sequence of elements sent from the source. These elements are termed LDUs (Logical Data Units) and contain the actual data (such as video frames). The emittances of events that LDUs are sent, are accumulated in an EventSequence `SE`. In addition, `Flow` defines an attribute called `initiate` that contains the event, if any, that started this flow.

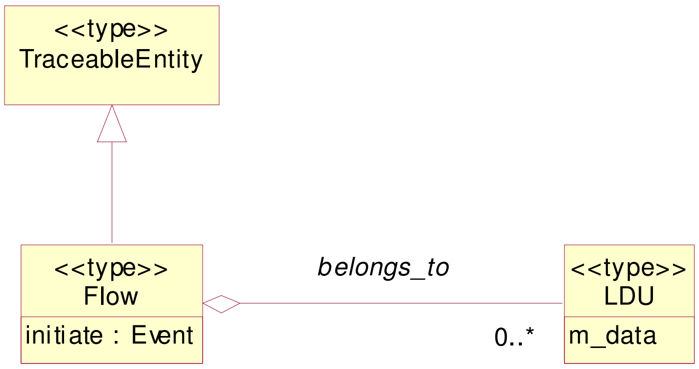


Figure 17: Model of Flow

With a `Flow`-object available as a parameter, the `startUpTime` characteristic can be defined as:

```
quality_characteristic startUpTime (flow : Flow) {
 domain: decreasing numeric milliseconds;
 values: if flow.SE->isEmpty then invalid
    else flow.SE->first.time() - flow.initiate.time()
    endif;
 invariant : flow.initiate = invalid implies flow.SE->isEmpty;
}
```

This definition includes a definition initiated by the keyword **`values`**. This part defines the values of this characteristic. In this case, values of the `startUpTime` characteristic are defined as: if the sequence of events of service emittances in the flow is not empty, the difference in time between the time of the first element of the flow and the time of the initiating event. In other words, `startUpTime` represents how long it takes for a flow to start to emit after a request to do so has been received. This definition also includes an invariant part specifying that if no initiating event exists, the flow does not contain any elements.

In the **`values`**-clause we exploit the fact from the computational model that objects have history by using this history to specify properties that the object must exhibit. We access the history by the `SE` and `SR` attributes as defined previously.

The parameter of the `startUpTime` characteristic is included because we need it when defining the values in order to relate some of its properties. If we were to define the values of `delay` or `output`, the objects that these characteristics are to be measured on would be needed as parameters. When discussing QoS specifications in subsection 4.6.5, we will further elaborate on parameters in the definition of QoS characteristics.

4.6.3.5 Composition

Components may consist of other components. These constituent components collectively supply the properties of the composite. Usually, composites contain collaborating components. We term such composites sequential since the properties of the composite are provided by components that interact sequentially. Sequential composition means there exist dependencies between the constituent components, for instance, a component may use services offered by another component. Composites can also be parallel. This is the case if no dependencies exist between the constituent components. Composition of several equivalent components is a parallel composition. For instance, a binding object may consist of two equivalent network connections, and the binding object may use any one of them to perform its services. Such parallel composites where the constituent components are equivalent have an or-relation between its constituents; one or the other may be used. The constituent components in parallel composites may also be of different types as long as they are unrelated. For example, a multimedia client may consist of a video window and a text window that are not related. Together these make up the client, but may otherwise be unrelated. Such a parallel composite has an and-relation between its constituent components since both components are needed for the composite to offer its services.

For any of the three types of compositions (sequential, parallel-and and parallel-or), a binary operator can be specified, i.e., an OCL expression with signature `QoSChar` × `QoSChar` → `QoSChar`, where `QoSChar` is the type of the QoS characteristic. The composition clause defines the result, with respect to the QoS characteristic under consideration, of composition of two components with the same QoS characteristic. Note that we specify the results of composition of components along one dimension (QoS characteristic) as a property of the QoS characteristic, and not as properties of the components. This can be done since evaluation of values of the QoS characteristic is based on properties of the components. When defining the compositional properties, only the values of the QoS characteristic under consideration of the two components to be composed are available in the OCL expression. This orthogonality is a restriction of the expressive power of CQML, one could imagine referring to other QoS characteristics of the components. However, this would lead to implicit bindings between the QoS characteristic specified and the QoS characteristics that are referred to since components having the QoS characteristic under specification would also need to have the QoS characteristics referred to in the definition of the composition. More importantly, by excluding cross-references between QoS characteristic definitions, we maintain independence of the QoS characteristics and avoid addressing feature interaction problems. Note that any feature interaction problems do not disappear; we only exclude them from what CQML supports.

To specify the effects of these different types of compositions, we extend the syntax of a QoS characteristic to incorporate composition:

```
<characteristic_definition>    ::= <domain_declaration> [<values_declaration>]
                                   [<composition_declaration>] [<invariant>]
<composition_declaration>      ::= composition ':' <composition_definition>
<composition_definition>       ::= <parallel_and> [<parallel_or>] [<sequential>]
                                |  <parallel_or> [<sequential>]
                                |  <sequential>
<parallel_and>                 ::= parallel-and <binary_expression> ';'
<parallel_or>                  ::= parallel-or <binary_expression> ';'
<sequential>                   ::= sequential <binary_expression> ';'
<binary_expression>            ::= <OCL::multiplicativeExpression>
                                |  <OCL::additiveExpression >
                                |  <OCL::featureCall>
```

For the `startUpTime` characteristic defined above, parallel or-composition results in the minimum start-up time, parallel and-composition results in the maximum start-up time, while a sequential composition results in the sum of the values. In order to use the two values that are being composed in specification of the effect of the composition, we introduce the special name `lhs` that denotes the left-hand side and `rhs` that denotes the right-hand side of the binary expression. The definition of the QoS characteristic can be extended to:

```
quality_characteristic composableStartUpTime (flow : Flow) {
  domain:  decreasing numeric milliseconds;
  values:  if flow.SE->isEmpty then Undefined
           else flow.SE->first.time() - flow.initiate.time()
           endif;
  composition:  parallel-and lhs.max(rhs);
                parallel-or lhs.min(rhs);
                sequential lhs + rhs;
}
```

After the composition-keyword, the effects of the three types of compositions are defined. The effects are defined using binary OCL operators and referring to the special names `self` and `rhs`. The values of the `composableStartUpTime` are numeric, so the predefined OCL `min`-operator returns the minimum of two numeric values while the `max`-operator return the maximum. The +-operator returns the sum of the two values in the composition. The two types of parallel compositions are specified as **`parallel-and`** and **`parallel-or`**.

4.6.3.6 Keywords

To define QoS characteristics, we have introduced the following keywords, separated by "|":

`quality_characteristic` | **`domain`** | **`values`** | **`composition`** | **`invariant`** | **`increasing`** | **`decreasing`** | **`numeric`** | **`set`** | **`enum`** | **`with`** | **`order`** | **`maximum`** | **`minimum`** | **`range`** | **`mean`** | **`variance`** | **`standard_deviation`** | **`percentile`** | **`moment`** | **`frequency`** | **`distribution`** | **`parallel-and`** | **`parallel-or`** | **`sequential`**

In addition, the keywords of OCL are used to define values of the QoS characteristics and to define semantics of composition.

4.6.4 QoS Statements

We now have the vocabulary to specify the QoS characteristics that are relevant for applications. However, we cannot yet specify restrictions of the values of these characteristics that can be related to the application. To alleviate this we need QoS statements. A QoS statement is used to constrain a set of QoS characteristics for them to collectively represent (parts of) the total QoS of a component. For instance, we have specified a characteristic named `delay`. We can use this to specify a QoS statement that constrains values of this characteristic to be within a certain limit. We can for instance say "`delay < 4`". Since we have defined `delay` to be measured in milliseconds (informally), we have here a QoS statement saying that the delay is less than four milliseconds. How to interpret this statement depends on the context. As mentioned in subsection 4.5.1, there are five types of QoS statements. If a QoS statement is used as a specification of what a system offers, it will be classified as a QoS offer, while if it is used to specify what a system expects, it is classified as a QoS expectation. The use of QoS statements is discussed in subsection 4.6.5, but first we define how to specify them.

A simple QoS statement can be defined using the following syntax:

<quality_declaration>	::=	**`quality`** <quality_name> '{' {<constraint> ';' }* '}'
<constraint>	::=	<simple_constraint> { <OCL::logicalOperator> <constraint> }*
	|	'(' <constraint> ')'
<simple_constraint>	::=	<characteristic_name> <OCL::relationalOperator> <value>
<value>	::=	<element> | <range_values> | <characteristic_name>

Given the `frameOutput` characteristic, a simple QoS statement called `guaranteed_high` can be specified as:

```
quality guaranteed_high {
  frameOutput >= 25;
}
```

This simple QoS statement defines that to always have a high output means to have an output of more than 25 frames per second.

QoS statements combine QoS characteristics and restrict their values. While the definition of a QoS characteristic identifies its domain and may restrict its values in this domain by an invariant, a QoS statement specifies value restrictions of each QoS characteristic that is included in the QoS statement.

To be able to use, in a QoS statement, QoS characteristics that have parameters as part of their definition, QoS statements must also have parameters. The parameters (or parts of these) of a QoS statement are passed on to the QoS characteristics; this means a QoS statement must have formal parameters from which all formal parameters for the QoS characteristics it uses can be extracted.

By default, QoS statements specify guaranteed constraints, i.e., constraints that must always be satisfied. If violations of such constraints occur, it is a matter of the QoS framework to handle the situation. CQML only provides the means to specify the constraints; it is the role of the QoS framework to enforce such constraints and handle violations if they occur. Since statistical derivations may be used, a guaranteed constraint may not be possible to refute until an appropriate number of measurements of its value have been made. How to identify a violation of a statistically derived QoS characteristic is a matter for the measurement mechanism. Alternatively to guaranteed constraints, best-effort constraints can be specified. Best-effort constraints indicate what QoS a component offers or requires, but no promises on the performance are made. Although not providing any absolute guarantees about the QoS levels, best-effort constraints can be used to specify what types of QoS characteristics a component provides or requires. This is useful if a client wants to adapt according to the prevailing conditions since it can examine the values of the QoS characteristics if the service provider offers inspecting operations. Without best-effort QoS statements, clients cannot know what QoS characteristics a service provider has.

Best-effort QoS statements may be qualified by **`threshold`** or **`compulsory`** accompanied with specification of a limit value. Threshold means that that the service will warn its clients if the provided QoS drops below the value specified as the limit. To be able to warn, a record must be kept of the clients of the service. To keep this record and to warn the clients is the responsibility of the underlying QoS framework. The QoS framework can for instance include code-generators that generate standard callback-handlers for the clients of a threshold best-effort service, that the QoS framework can use to notify of a violation, possibly by using a standard notification service.

`compulsory` is the other best-effort qualifier. As opposed to a threshold service, **`compulsory`** specifies that the service will abort if unable to maintain the specified level of QoS. Also in this case, it is the responsibility of the QoS framework to notify the clients if a service breakdown occurs.

We can specify best-effort QoS statements and its different types as:

```
<constraint>            ::=  <simple_constraint> | <qualified_constraint>
<simple_constraint>     ::=  <single_constraint>
                             { <OCL::logicalOperator> <simple_constraint> }*
                        |    '(' <simple_constraint> ')'
<single_constraint>     ::=  <characteristic_name> <OCL::relationalOperator> <value>
```

```
<qualified_constraint>   ::= guaranteed <simple_constraint>
                          |  best-effort <simple_constraint>
                          |  <qualifier> best-effort <single_constraint>
                             with limit <element>
<qualifier>              ::= threshold | compulsory
```

Note that since a QoS statement by default is guaranteed, the **guaranteed**-keyword is only included to enable modellers to show explicitly that a QoS statement is guaranteed. If best-effort is to be specified, the keyword **best-effort** is used:

```
quality try_high {
  best-effort frameOutput > 25;
}
```

If a compulsory or threshold best-effort QoS is to be specified, we use corresponding keywords with accompanying specification of the threshold or compulsory levels:

```
quality high {
  compulsory best-effort frameOutput > 25 with limit 20;
  threshold best-effort delay < 5 with limit 8;
}
```

Note that qualified best-effort QoS statements can only specify single constraints since they must specify limit values. To use compound QoS statements for this would require the limit value specification to be correspondingly complex; this is a possible future extension to CQML.

More complex QoS statements can combine a number of QoS characteristics, some of which may potentially be statistical characteristics:

```
quality normallyFast {
  frameOutput > 25;
  statisticalDelay.maximum < 50;
  statisticalDelay.minimum > 1;  //albeit an unnatural
                                 //specification
  statisticalDelay.range = (1..50); //follows from maximum < 50
                                    //and minimum > 1
  statisticalDelay.mean < 20;
  statisticalDelay.variance < 3;
  statisticalDelay.standard_deviation < 9;
  statisticalDelay.percentile 90 < 3;
  statisticalDelay.percentile 50 < 12;
  statisticalDelay.moment 3 < 1;
  statisticalDelay.frequency 10 < 15;
}
```

We have included all statistical aspects for illustration purposes only, normally only a subset of these is used. A more useful QoS statement is:

```
quality fast {
  frameOutput.mean > 25;
  frameOutput.minimum > 20;
  delay.maximum < 50;
  delay.mean < 10;
}
```

When a number of QoS characteristics with their restrictions are included in a QoS statement, all restrictions apply (i.e., the restrictions are and-ed). CQML also includes the possibility to have an or-relationship between restrictions to be able to specify more complex QoS statements:

```
quality fast_alternatives {
  (frameOutput.mean > 25 and frameOutput.minimum > 20) or
     (frameOutput.mean > 30 and frameOutput.minimum > 10);
  delay.maximum < 50;
  delay.mean < 10;
}
```

Note that since output is an increasing domain, it is often useful to define its minimum, and for delay being a decreasing domain, it can be useful to specify its maximum. Also note that we have here used QoS characteristics that were not initially defined with statistical derivations (`frameOutput` and `delay`), but nevertheless added such derivations in the definition of the QoS statement. We can add such derivations whenever the domain of the QoS characteristic allows.

Until now, we have only restricted the values of some simple characteristics in the QoS statements. However, to support reuse of specifications, QoS statements may be specialised. To be a specialisation means that it must restrict its base type and it can do so by adding new constrained QoS characteristics or restricting the values of the QoS characteristics in the base further. Furthermore, since parameters to the QoS statement may be collections containing objects that each has QoS characteristics that are to be constrained, we allow the use of OCL expressions when defining the constraints. Hence, to allow specialisation, to include the ability to have parameters, and to allow OCL expressions as part of the constraint specification, the grammar is extended to:

```
<quality_declaration>      ::= quality <quality_name>
                               ['(' <OCL::formalParameterList> ')']
                               [<quality_specialisation>] '{' <constraint>* '}'
<quality_specialisation>   ::= ':' <quality_name> ['(' <OCL::actualParameterList> ')' ]
<constraint>               ::= <simple_constraint> | <qualified_constraint>
<simple_constraint>        ::= <single_constraint>
                               { <OCL::logicalOperator> <single_constraint> }*
                               | '(' <single_constraint> ')'
                               | <OCL::logicalExpression>
<single_constraint>        ::= <characteristic_name> ['(' <OCL::actualParameterList) ')' ]
                               ['.' <one_aspect> ] <OCL::relationalOperator> <value>
```

Note that collections as parameters for the QoS statement in many cases are results of compositions, and the composition specification on the parts therefore defines the QoS characteristic to be used in constraints. Note also that we have included statistical aspects in the single constraint.

The defined characteristic `startUpTime` included parameters in its definition, and we need to include these parameters in the QoS statements:

```
quality quickAndAlert (flow : Flow) : fast {
  startUpTime(flow) < 100;
}
```

The definition of `quickAndAlert` inherits the properties of `frameOutput` and `delay` from `fast`. It then adds a restriction of the `startUpTime` to be less than 100 milliseconds. Since `startUpTime` needs parameters in its definition, `quickAndAlert` needs to be able to extract those parameters from its own parameters. The definition of `delay` and `frameOutput` would also benefit from including a parameter so that the semantics of these characteristics can be specified; we include this in the discussion in subsubsection 4.6.5.2.

To illustrate a QoS statement that constrains a collection, we assume `Stream` to be a collection of `Flows`. The `quickAndAlert` QoS statement can be defined as:

```
quality quickAndAlert (s : Stream) : fast {
  s->forAll(f : Flow | startUpTime(f) < 100);
}
```

A QoS statement restricts a number of QoS characteristics within some bounds, and as such represents a possible level of QoS. Hence, a QoS statement can be viewed as a predicate defining, when evaluated, whether some QoS observations are within the level of QoS or not. [Loyall et al., 1998] terms this level of QoS as a region in their system Quality Objects (QuO).

4.6.4.1 Keywords

We have introduced the following keywords to define QoS statements, separated by '`|`':

`quality` `|` **`guaranteed`** `|` **`best-effort`** `|` **`threshold`** `|` **`compulsory`** `|` **`with limit`**

In addition, the operators of OCL are used to define bounds of the QoS characteristics.

4.6.5 QoS Profiles

We now have the vocabulary available to specify QoS statements, but we are still not able to bind a specific QoS to a component in a system. There are four possible approaches to address this:

1. Keep whatever IDL is used to specify the component's functional properties unchanged, and define QoS parameters as types in IDL that can be used when defining component interfaces. This is the approach used in the specification of CORBA audio/video streams [OMG, 1998], where QoS is defined as a sequence of name/value pairs used in interface definitions.
2. Implicitly enhance the chosen IDL to include QoS. For instance, this may be done by introducing keywords in comments that can be parsed by an extended IDL parser or by introducing special IDL identifiers that are treated differently to regular IDL identifiers by an extended IDL parser. TAO [Schmidt et al., 1998] uses this last approach by introducing the `RT_Operation` interface in CORBA IDL that is used to convey QoS information to the ORB for CPU scheduling of real-time IDL operations.
3. Explicitly extend the chosen IDL by introducing new keywords. This is the approach of MAQS [Becker and Geihs, 1997] where they extend CORBA IDL to include the two new keywords `qos` and `withQoS`.
4. Attach QoS to components separately from the definition of their functional properties. This is the approach of QML [Frølund and Koistinen, 1998a] and QuO [Loyall et al., 1998] where they use IDL as is, and introduce new languages to specify QoS and to bind the specified QoS to a component.

The requirement of separation of QoS specification and interface specification precludes all but the last approach, which we adopt.

We use CIDL (the Component Interface Definition Language) [OMG, 1999a] to illustrate specification of QoS of components. CIDL is currently being defined by OMG to specify components. CQML is not tied to CIDL; CIDL is used for illustration purposes only.

4.6.5.1 Introduction to the Example

In order to illustrate the use of CQML, we use a simplified video camera design based on an example from [Blair and Stefani, 1997]. The example is detailed in subsection 4.8.1, in the following we only make use of some parts to illustrate the concepts. The example consists of a video camera that emits a video flow and a video window that presents the flow to the user.

Between these two, a binding object is responsible for transferring all relevant information in a timely manner. Figure 18 shows the basic set-up of this example.

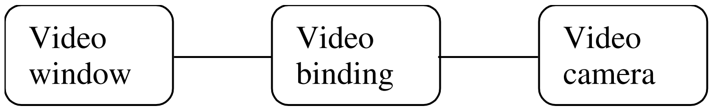


Figure 18: Video application set-up

A system analyst would only specify the video camera and the video window with a logical connection between them. The video camera provides a service (a video flow) with a certain QoS offer. In addition, the video window provides a service to the end user (showing the video) with a certain QoS offer given that it is fed video information with appropriate quality. Hence, a system analyst would specify the QoS offers and QoS expectations of these analysis artifacts. A system designer would be the one to introduce a binding object. Based on the QoS offers and QoS expectations of the analysis artifacts, a system designer would then specify appropriate QoS offers and QoS expectations for the design artifacts (the video binding in this case).

When introducing QoS offers and QoS expectations next, we define interfaces of these components using CORBA IDL, specify components that provide these interfaces using CIDL, and relate QoS to the components using CQML. Based on this information, a service broker could link components together in an end-to-end service. The essential part of the example is that the video camera captures a sequence of video frames that it sends through an interface termed `VideoStream`. This sequence of video frames (termed "flow" due to its continuous nature) is transferred by the `videoBinding` to the `videoWindow` that presents it. The frames the camera emits should be transferred by the binding and presented to the end user without too much delay.

4.6.5.2 QoS Offers and QoS Expectations

The video stream interface, through which the video frames are sent, can be defined as:

```
interface VideoStream {
  Video videoFlow;
};
```

We assume a predefined type `Video` that represents a continuous flow of video frames. The interface definition specifies that any object that provides this interface must have an object of type `Video` available for the clients to use to access the continuous flow of video frames the object produces. This interface can be implemented by a number of objects offering different QoS. To specify an object to implement this interface, we define a component template `myFastCamera` using CIDL:

```
component myFastCamera {
  provides VideoStream outgoing;
};
```

A component adhering to this specification provides access to its provided interface `VideoStream` through the name `outgoing`.

We are now in the position to specify the QoS for `myFastCamera`. We adopt the approach of QML to specify QoS profiles that associate QoS statements with component specifications or parts thereof. QoS profiles are associated with components (i.e., objects). Such a QoS profile defines properties of the component both with respect to what QoS it requires from the environment and what QoS it offers. The properties defined in a QoS profile define a dependant type. Such a type depends on the component with which it is associated. The profile restricts the state and behaviour of the component further from its initial definition, and the component definition with an associated profile represents as such a subtype of the component type. A component type may have several associated profiles representing different subtypes.

The syntax of QoS profiles is:

<profile>	::=	**profile** <profile_name> **for** <component_name> '{' <profile_body> '}'
<profile_body>	::=	<expectation_specification> [<offer_specification>]
	|	<offer_specification>
<expectation_specification>	::=	**uses** <expectation> ';'
<offer_specification>	::=	**provides** <offer> ';'
<expectation>	::=	<expectation_name> ['(' <OCL::actualParameterList> ')'] {<OCL::logicalOperator> <expectation> }*
	|	'(' <expectation> ')'
<offer>	::=	<offer_name> ['(' <OCL::actualParameter> ')'] {<OCL::logicalOperator> <offer> }*
	|	'(' <offer> ')'

A profile that associates the QoS statement `high` with the component specification `myFastCamera` can be specified as:

```
profile goodCamera for myFastCamera {
  provides high;
}
```

Component implementations following this profile need to provide the `high` QoS. However, we have not specified the semantics of this statement. From the definition of `high` in subsection 4.6.4, we see that the `myFastCamera` component should provide the `frameOutput` above 25, but there is no direct relationship between the component `myFastCamera` and the `frameOutput` property used in the `high` QoS specification. To make this connection, we may require that the `myFastCamera` object has the property `frameOutput` and this has numeric values. This property can either be an attribute in the definition of the interface, or one can use a property service as defined in CORBA to define it.

However, the requirement that the component needs to have a certain set of properties (e.g., `frameOutput`) is in many cases too restrictive. Systems exist that were not designed with QoS in mind, but one may nevertheless want to specify QoS at a later stage. For many such systems, to extend the interfaces with new properties is not an option. We therefore introduce the possibility of specifying QoS statements with parameters that can be used when specifying QoS. In order to illustrate this, we introduce refinements of the QoS characteristics and QoS statements used in the example first.

To simplify the QoS statements in this example, we define utility operations in OCL on the `EventSequence` type introduced earlier:

```
context EventSequence::backInTime(
                range : Integer) : EventSequence
  post: if EventSequence->last.time() -
            EventSequence->first.time() < range
        then result = Undefined
        else result = EventSequence->iterate(
          event : Event; acc : EventSequence = Sequence{} |
                if EventSequence->last.time() - event.time()
                                                    <= range
                then acc.append(event)
                else acc
                endif )
        endif
```

This operation returns the subsequence of events that occurred within the interval from the last event and backwards a number of milliseconds defined by the `range` parameter.

```
context EventSequence::eventsInRange(
          milliseconds : Integer) : Integer
  post: result = EventSequence->backInTime(milliseconds)->size
```

The result of this operation is the number of events a sequence contains from the last event and backwards the number of milliseconds specified by the `milliseconds` parameter.

We then redefine the QoS characteristic `frameOutput` as:

```
quality_characteristic frameOutput (flow : Flow) {
  domain: increasing numeric frames/second;
  values: flow.SE->eventsInRange(1000);
}
```

This definition of the QoS characteristic specifies the values to be the number of events of a frame being sent within the last second, counted from when the last frame was sent. The value of this characteristic is dynamic since it varies as time goes. Note that if the flow ends, the value of this characteristic would still be the last number of frames sent. If the semantics were to be the last second from the time of invocation, the `EventSequence.last.time()` statements in the definition of `backInTime` should be replaced with the time of invocation.

Based on this new definition of `frameOutput`, the `high` QoS statement can be specified as:

```
quality high (flow : Flow) {
  frameOutput(flow) >= 25;
}
```

This QoS statement can then be used when defining the QoS profile of the video camera, as the `videoOut` object is of type `video` that is one subtype of `Flow` as used in the definitions above:

```
profile goodCamera for myFastCamera {
  provides high(outgoing.videoFlow);
}
```

We have now established a semantic link between the video flow represented by the `videoFlow` object in the interface named `outgoing` and the quality it is expected to provide. We have specified that the sequence of video frame emittances at any time contains at least 25 occurrences within the last second.

In terms of aspect-oriented programming [Kiczales et al., 1997], a QoS profile specifies aspects of the component. That is, it cross-cuts the component with properties that cannot be encapsulated as functionality in a single method. To support a QoS profile, a component must in most cases have QoS support embodied into most parts of its implementation.

We have specified the QoS offer for a component that provides a service without being dependent on other services in its environment. In terms of a QoS relation as discussed in subsection 4.2.1, it is a pure obligation that can be expressed as

```
    TRUE ↦ high(myFastCamera.outgoing.videoFlow)
```

In most cases, the expectation is not trivially true. For instance, in our example, the video-binding needs to be fed with a flow of a certain quality in order to be able to transport it to the consumer (video window). As discussed in subsection 4.5.5, such a binding has at least two interfaces corresponding to the interfaces of the client and the server. In the case of the video binding, the interfaces are similar to the interface of the video camera, and only distinguished by the direction of the flow, i.e., whether the interface is incoming or outgoing.

A main QoS characteristic of the binding is its delay. We can redefine the `fast` QoS statement to include only the delay characteristic, but must then first redefine the delay characteristic to include parameters. To do this, we define a utility operation `maxTimeDifference` on the `EventSequence` type:

```
context EventSequence::maxTimeDifference(
          otherSeq : EventSequence) : Integer
  post: EventSequence->iterate(
          firstEvent : Event;
          max : Integer = Undefined |
            let time_difference : Integer =
                (otherSeq->select(secondEvent : Event |
                  secondEvent = firstEvent)->first.time() -
                    firstEvent.time()).abs in
            if (time_difference > max) or (max = Undefined)
            then time_difference
            else max
            endif
        )
```

The result of this operation is the maximum time difference between any two "equal" events in two event sequences (`secondEvent = firstEvent`). However, since events are unique we cannot use a standard equality operation based on identity to compare two different events; they would always be different. We therefore use the equality operator introduced in Figure 15 that compares two events by checking their context. That is, we use frame identity to ensure that "equal" means that the events are related to the same video frame. Based on this, we can identify the two corresponding events of emitting a video frame and receiving acknowledgement of its reception, and then compute the maximum time difference between any two such events in the two event sequences of receiving and emitting video frames. Using this operation, delay can be defined as half of the round-trip delay:

```
quality_characteristic delay(outFlow : Flow) {
  domain: decreasing numeric milliseconds;
  values: outFlow.SE->maxTimeDifference(outFlow.SR) / 2;
}
```

We here use the fact that a `Flow` accumulates the events of frame emittances in the `SE`-sequence, and the acknowledgement of its reception in the `SR`-sequence as an ordinary interrogation. With this addition of values to the `delay` characteristic, we can redefine the `fast` QoS statement:

```
quality fast (outFlow : Flow) {
  delay(outFlow) < 10;
}
```

The `videoBinding` component can be specified as:

```
component videoBinding {
  uses VideoStream incoming;
  provides VideoStream outgoing;
};
```

We have here specified that the binding object supports both the incoming and outgoing interfaces in order to transport the flow. A corresponding QoS profile can then be specified as:

```
profile goodBinding for videoBinding {
  uses high(incoming.videoFlow);
  provides high(outgoing.videoFlow) and
           fast(outgoing.videoFlow);
}
```

We have here specified that, if it gets a flow with high frame rate, it delivers a flow with a high frame rate with delay less than 10 milliseconds. The first part is the QoS expectation while the second part represents the QoS offer. The specification of the `goodBinding` above corresponds to the QoS relation:

```
videoBinding: high(incoming.videoFlow) ↦
     high(outgoing.videoFlow) and fast(outgoing.videoFlow)
```

QoS expectations in QoS profiles are specified after the **`uses`**-keyword, while QoS offers are specified after the **`provides`**-keyword. The **`provides`**- and **`uses`**-clauses are logical expressions of QoS statements. Any parameters used in the QoS statements must be elements of the component, ensuring that the QoS characteristics used in the QoS statements indeed are aspects that can be evaluated from the component properties.

QoS profiles may also be specified as specialisations of other profiles. The syntax is extended to:

<profile>	::=	**`profile`** <profile_name> [<profile_specialisation>] **`for`** <component_name> '`{`'<profile_body> '`}`'
<profile_specialisation>	::=	'`:`' <profile_name>

A specialised profile restricts its base, and it does so by extending its QoS offer or by slackening the restrictions it poses on the environment. To extend the QoS offer means to add QoS statements in the provides-clause of the derived QoS profile; these are added in a conjunction with the QoS offer in the base in order to strengthen it. To slacken the restrictions on the environment also means to add QoS statements in the uses-clause of the derived QoS profile, but these are added in a disjunction with the QoS expectation in the base in order to relax it.

4.6.5.3 QoS Contracts and Adaptation

We have until now focussed on individual components by making possible the specification of QoS offers and QoS expectations. However, components interact to collectively provide the required system behaviour. The focus on individual components does not cover such interactions sufficiently; we need to link what one component offers with what another expects for them to be able to collaborate. A QoS contract is an agreement at run-time between collaborating components on what QoS each component is responsible to provide. However, it is important to note that CQML does not implement adaptation or the agreement process for establishing QoS contracts, CQML only facilitates the QoS framework in doing so by providing the system developers appropriate concepts to use when specifying component properties. The QoS framework can then use these properties for establishment and enforcement of QoS contracts and for adaptation at run-time.

QoS expectations of a component are not bound to particular components in the environment; they only refer to what the component expects from the environment. Furthermore, a component specification may have many associated QoS profiles. However, for any one instance of a collaboration between two component instances, only one QoS profile is used for each of the two component instances, and these two QoS profiles constitute a QoS contract. These two QoS profiles must match partially, i.e., parts of what one requires must be provided by the other and sometimes vice versa. It is therefore important to have a precise definition of what it means for two profiles in CQML to match partially; this is addressed in subsection 4.7.3.

Two components can collaborate if parts of what is offered by one conform to parts of what is expected by the other. A service provider may be used even if it only provides some of the services a client needs; the client may turn to other service providers to meet its other requirements. In addition to conformance of the functional aspects, also the QoS constraints must be matched. A match between QoS profiles means that the uses-clause of one conforms to the provides-clause of some others, i.e., they offer at least the same QoS properties that are required. However, since only a partial match is required for a collaboration to take place, the uses-clause of the client may only be partially matched by the provides-clause of the service provider in order for two components to collaborate. Sometimes it is also required that the client provides some services in return for the collaboration to take place, e.g., when callback handlers are required. In such cases, it is important that the two collaborating partners are identified; it does not make sense for a component to offer callback handlers to another instance than the one it uses services from. If such identity constraints apply, both the uses- and the provides-clauses for both components must be part of a contract. In order to establish such contracts, each party is only concerned with its side, i.e., that it gets the services it needs. If a party, as part of getting to an agreement, has to provide some services in return, this request must be initiated by the other party. Conditional service offers where part of the conditions is that the collaborating component must provide matching services, are specified in the component definition language. However, similar constraints may apply to QoS profiles so that one can specify the identity of providers of some services with certain qualities must correspond to the identity of a service user. In CQML, this is supported by the possibility to specify invariants in QoS profiles and QoS statements. Provided an identity operation is available, this enables the modeller to specify that the identities of the components to which some of the properties apply, must be equal. Invariants are added to QoS profiles as:

```
<profile_body> ::=        {<expectation> [ <offer> ] | <offer>} [<invariant>]
```

The computational model presented in section 4.5 supports object identity, and since interfaces and parts thereof pertain to an object, object identity can be derived from any part used in a QoS profile. To provide an identity operation, a similar approach to the event context can be supported. However, it is outside the scope of CQML to address this; we use OCL for this.

Since types are predicates characterising a collection of objects, profile matching corresponds to constraint satisfaction under the restriction of functional conformance. The constraint satisfaction problem is well-known and studied since the early seventies (see for instance [Tsang, 1993, Ruttkay, 1998]), and use of results from this field of study would be useful in a QoS framework. An important result from the research on constraint satisfaction is that constraint optimisation is NP-complete [Mackworth, 1977]. Hence, search for the single optimal QoS offer in a negotiation should use heuristics when proposing new QoS offers, or one should settle for any one QoS profile that satisfies the expectations of the client.

If it is not possible to match the offered and expected characteristics even if their functional properties are conformant, strengthened offers or relaxed expectations must be put forward in order to establish a QoS contract. Hence, to establish a QoS contract may require negotiation. This requires that the components are able to adapt, i.e., change their behaviour.

QoS specifications can be used to match offers and expectations at design-time to compose an application from individual components. However, QoS specifications only reflect the static, design-time view of the components. At run-time, system conditions may vary. Preferably, variations should be handled by the underlying infrastructure or ignored if they are only minor so that the application components remain unaffected. Alternatively, the application components themselves may have implemented strategies to handle varying system conditions without exposing changes of their provided QoS. In both cases, the variations in system conditions are invisible from the outside of a component. In [Chalmers and Sloman, 1999], this black-box adaptation is termed maintenance and defined to be “modification of parameters by the system to maintain QoS. Applications are not required to modify behavior".

However, some variations are too severe to be handled transparently. In such cases, assumptions made during design-time will no longer be valid and the component must either choose to adapt, to continue unaffected and thereby expose the immediate effects of the new conditions, or to simply abort. To simply abort or to continue unchanged are the straightforward alternatives, but are in most cases not acceptable. It is often desirable to continue the service at another QoS, but measures should be taken to alleviate the effects of the system variations. In [Chalmers and Sloman, 1999], this is termed adaptation and defined to be "when the applications adapt to changes in the QoS of the system". Note that what is referred to as maintenance at one level of abstraction is indeed adaptation at a lower level of abstraction. Hence, the distinction between maintenance and adaptation is relative to the level of abstraction.

For a component to be able to adapt, it must realise when to adapt, there must exist alternative behaviours to use when variations occur, and it must be possible to decide which alternative behaviour to use. QoS expectations are used to decide when to adapt. If the expectation of a component is no longer met, adaptation is needed. Hence, if `Exp(O)` denotes the default QoS expectation of a component `O`, the QoS relation for that component can be extended to

$$(\mathtt{Exp(O)} \mapsto \mathtt{Obl(O)}) \wedge (\forall \mathtt{i} \mid \mathtt{Exp}_\mathtt{i}\mathtt{(O)} \mapsto \mathtt{Obl}_\mathtt{i}\mathtt{(O)})$$

where the additional expectations $\mathtt{Exp}_\mathtt{i}\mathtt{(O)}$ represents possible alternative expectations and the corresponding obligation $\mathtt{Obl}_\mathtt{i}\mathtt{(O)}$ the behaviour of the component in such cases, and where $\forall \mathtt{i} \mid \mathtt{Exp(O)} \mapsto \mathtt{Exp}_\mathtt{i}\mathtt{(O)}$, i.e., all alternative expectations are relaxed constraints on the environment.

The objective of adaptation is to maintain the level of QoS as far as possible. One approach to accomplish this is to specify a number of QoS profiles for the components. If system conditions change to invalidate the default QoS expectation of a component, other QoS profiles may have QoS expectations that still hold. If the list of QoS profiles is ordered, the "best" QoS profile with valid expectations can be chosen and a new QoS contract can be established. When the component changes QoS offer (i.e., invalidates the current QoS expectation and chooses a new QoS profile), behaviour can be triggered. Such behaviour may for instance be used to notify the components involved about the altered conditions. This is the approach used in QuO where individual QoS offers are termed regions and region transitions may trigger callback handlers, and is parallel to what in [Waddington and Hutchison, 1998] is called "resource capacity regions". This approach does not require components to be able to provide new QoS offers dynamically since if the set of QoS offers are specified at design time, adaptation only involves finding what QoS offers are valid and re-configuring the application accordingly.

An alternative approach is for components to be able to put forward new QoS profiles dynamically. This approach acknowledges the fact that one cannot always predict the possible conditions of the environment. The QoS relation of a component `O` can in this case be specified as:

$$(\mathtt{Exp(O)} \mapsto \mathtt{Obl(O)}) \wedge (\neg\mathtt{Exp(O)} \Rightarrow (\exists \mathtt{i} \mid \mathtt{Exp}_\mathtt{i}\mathtt{(O)} \mapsto \mathtt{Obl}_\mathtt{i}\mathtt{(O)})$$

with `Exp(O)` denoting the default expectation and $\neg$`Exp(O)` denoting a violation of this. The QoS relation specifies that if the default expectation is violated, there exists an expectation such that an obligation is met. Such an obligation is often termed an exception, e.g., in CORBA and Java computational models.

Using the approach of being able to dynamically put forward new QoS profiles, to establish new QoS contracts involves re-negotiation based on the new profiles, not only based on design-time specification of potential QoS profiles. In [Koistinen and Seetharaman, 1998], they describe a model for a QoS negotiation mechanism that presupposes this approach. It handles negotiation involving multiple offers and counter offers being made available at run-time. Their approach applies worth calculation of the QoS offers that enable agreements where the offers and expectations are only partially satisfied. When a client requests a service, it gives different worth values to elements in the expectation. Based on this, the server can create an offer that suits the client. If no suitable offer can be made, the server may compute a counter offer to the client. This presupposes that QoS offers not known at design-time can be put forward as part of the negotiation protocol and that elements in the expectations can be given worth values.

Both approaches to adaptation are useful and are supported by CQML. The static approach of specifying a number of QoS offers at design-time is effective at run-time since if adaptation is needed, the ordered list of QoS profiles is used to find the appropriate QoS offer and there is no need to engage in calculation of new offers and conveyance of these between client and service providers. On the other hand, the dynamic approach of specifying new QoS offers at run-time is flexible as the components may introduce QoS offers not possible to foresee at design-time. This is especially useful if worth-based negotiation is used since clients may convey their worth values to the service provider and the service provider can then adapt accordingly. The two approaches may be viewed as analogous to the static and dynamic invocation interfaces in CORBA.

To support the static approach to adaptation, we extend CQML to be able to specify a number of ordered QoS profiles. We adopt the approach of named regions with explicit transitions from QuO, renaming them to profiles to coincide with our terminology. We extend the syntax of QoS profiles to:

```
<profile>                     ::= profile <profile_name> [<profile_specialisation>] for
                                  <component_name> '{' [ <profile_body> ] '}'
<profile_body>                ::= <single_profile_body>
                              |   <one_profile>+ <transition>+ [ <transition_specification> ]
<single_profile_body>         ::= { <expectation_specification> [ <offer_specification> ]
                                    | <offer_specification> } [<invariant>]
<one_profile>                 ::= profile <profile_name> '{' <single_profile_body> '}'
<transition>                  ::= transition <transition_profiles> ':' <transition_body>
<transition_profiles>         ::= {<profile_name> | any } '->' <profile_name>
                              |   <profile_name> '->' any
<transition_body>             ::= <OCL::featureCall> ';'
                              |   '{' {<OCL::featureCall> ';'}+ '}'
<transition_specification>    ::= <precedence>
<precedence>                  ::= precedence <profile_name> { ',' <profile_name> }+ ';'
```

Given appropriate QoS statements and interface definitions, a QoS profile for a video binding can be specified as:

```
profile adaptiveBinding for videoBinding {
  profile good {
    uses high(incoming.videoFlow);
    provides high(outgoing.videoFlow) and
               fast(outgoing.videoFlow);
  }
  profile average {
    uses medium(incoming.videoFlow);
    provides medium(outgoing.videoFlow) and
               fast(outgoing.videoFlow);
  }
  transition any->average: incoming.some_callback("average");
  transition average->good: {
     incoming.some_callback("good");
     outgoing.another_callback("good");
  }
  precedence: good, average;
}
```

Two profiles named `good` and `average` with different QoS offers and QoS expectations are defined, and behaviour on transitions between them is defined as invocations of callback operations. The callback handlers have in this example only string parameters to convey information about why the callbacks are being invoked, but they may be defined to have parameters that represent the relevant parts of the state of the component to give the receiver explicit information about the state of affairs. The profiles are ordered in the precedence-clause, the default ordering is the order in which the profiles are listed. In the current version of CQML, the precedence ordering is the only way to specify which profile to chose when the component is initialised or when a profile transition occurs. However, the grammar includes a placeholder for more elaborate mechanisms such as state machines as future extensions.

Note that we have specified two callback handlers for the transition from the `average` profile to the `good` profile, one for the `incoming` and one for the `outgoing` interface. This corresponds to callbacks to the server and the client, respectively. Note also that we have used the keyword **`any`** to specify that a transition from any profile to `average` triggers `incoming.some_callback("average")`. In this case, however, only two profiles are specified and the use of **`any`** is included for illustration purposes only.

It is the responsibility of the QoS framework to invoke the appropriate operations whenever a profile transition occurs. In CQML, it is only specified what operations to invoke and under what circumstances to invoke them, but the actual invocations at run-time are done by the QoS framework based on the information specified in CQML and the measured current state of affairs. The different profiles, transitions and precedences are specified by the application developer and requires intimate knowledge about how the components are able to react to changes in their environment.

In a QoS profile specified as a sequence of named profiles, each such profile represents a possible QoS offer with its accompanying QoS expectation. To deterministically determine which of the profiles a component provides at any point in time when using a precedence-list, the sequence of profiles must be totally ordered. This ordering is computed based on the order of specification (implicit ordering) and by the list in the precedence-clause (explicit ordering), if any. Using OCL-sequences in the pseudocode, the total order of the profiles can be computed as follows:

```
//Use the ordering in the precedence-clause as starting point
Sequence total_order := explicit_order;
// Run through all named profiles from start
for i:=0 to implicit_order->size - 1 {
  if not(total_order->contains(implicit_order->at(i))) {
  // Not part of explicit ordering
    if i = 0 {
    //First profile and not explicitly ordered
      total_order->prepend(implicit_order->at(i));
    }
```

```
      else {
        for j:=total_order->size-1 to 0 {
        //Run through all explicitly ordered profiles
          if total_order->at(j) = implicit_order->at(i-1)
            break; // Found previous profile; stop search
        }
        // Add the new profile
        ((total_order->subSequence(0,j)
          )->append(implicit_order->at(i))
            )->union(total_order->subSequence
                                          (j,total_order->size));
      }
    }
}
```

Using the algorithm above, each profile that is not part of the explicit order is inserted into the total order immediately after the last profile in the total order that precedes it in the implicit order. Note that the ordering does not imply that the profiles are ordered according to the value of their offers for a client or according to how easy it is for the environment to satisfy the QoS expectations; any ordering is possible to specify. Since the profiles may include QoS statements that address different QoS characteristics, an ordering only based on the QoS offers or QoS expectations would, in such cases, not be total. However, this "strongness" intuition of the ordering can be achieved with an appropriate explicit ordering or by defining the profiles in the appropriate order for the implicit ordering. Note that any order specified explicitly has precedence over the implicit order.

At any point in time, the first profile in the ordering that has a QoS expectation that can be met by the environment, determines the QoS offer of the component. When the environment of a component changes, transition between its profiles may occur. If the environment improves sufficiently (satisfies a QoS expectation earlier in the sequence of profiles), it satisfies the QoS expectations of a "better" profile (one ordered before the current) and a transition to this profile may occur. On the other hand, if the environment degrades, it may no longer be able to meet the current QoS expectation and a transition to a "worse" profile (one ordered after the current) may occur. This new profile must have relaxed constraints that the environment can meet. Note that, following this scheme, a profile with stronger constraints on the environment than any one of its predecessors will never be chosen as the profile for the component since it needs an improved environment to become active, but any transition will then go to a previous profile. It is the responsibility of the modeller to avoid such unreachable profiles. This transition scheme can be made more general to include possible transitions using a state machine based on the state of the environment. Such an approach enables transitions from a profile to several profiles depending on what part of the expectation that was violated. This is a topic for future extension.

When a transition between profiles occurs, transitional behaviour may be triggered. All transitions that trigger behaviour must be explicitly identified as a pair of profile names or by using the **any**-keyword. In such a pair of profile names, the first name denotes the profile that has been invalidated and the second name denotes the new profile that is to become active. The **any**-keyword denotes all profiles. If more than one transition apply, the transition that is specified first of the applicable transitions, is chosen. That several transitions apply can be the case if the **any**-keyword is used or if more than one transition is specified for the same pair of profiles. The behaviour to be triggered is specified in the transition-clause as methods to invoke. The methods must be part of the incoming or outgoing interfaces of the component and must be invoked by the supporting infrastructure.

If no profile has QoS expectations that the environment can meet, the component no longer provides any specific QoS offer. How this situation is handled depends on the policies and capabilities of the supporting infrastructure. The component can continue to provide its service with no specific QoS, it can stop providing its service, or it can participate in negotiation. The first two approaches represent naîve solutions, while negotiation requires substantial support from the infrastructure. One element of this support is the ability of the clients to calculate worth values so that the service provider can discover how an offer may affect its clients. It is the responsibility of the supporting infrastructure to implement negotiation support, including worth value calculation.

To support the dynamic approach with worth-based negotiation we need to specify features to enable the calculation of worth values. Following the approach of [Koistinen and Seetharaman, 1998], a QoS expectation may have a worth profile that is used to calculate one worth value of any given QoS offer. The objective is to be able to assess whether one QoS offer is better than another, as seen from the client side. The worth value is calculated by the worth of some of the constituents of the QoS offer. One possible scheme is as follows:

> Only those parts of a QoS offer used by the client may be subject to worth calculation, i.e., the parts that correspond to the outgoing interfaces of the client. A QoS offer contains QoS statements linked with logical connectives. Each QoS statement contains a number of constrained QoS characteristics. The worth values of QoS characteristics are calculated by worth functions (termed benefit function by [Chatterjee et al., 1998]). A worth function for a QoS characteristic is an arbitrarily complex function defined over the domain of the QoS characteristic and results in a numerical worth value:
>
> ```
> CharacteristicWorthValue : numerical =
>         WorthFunction (CharacteristicValue : Domain)
> ```
>
> The worth value of a constrained QoS characteristic is defined to be the maximum value of the worth function for QoS characteristic values satisfying the constraint. Worth functions of QoS characteristics with statistical constraints need more complex functions than maximum to calculate the worth value.
>
> Since a QoS statement may include any number of QoS characteristics, worth values must be given to them so that the compound worth value of the QoS statement can be calculated. The worth value of a QoS statement specified in conjunctive normal form is the product of the individual worth values of the conjuncts:
>
> $$\mathtt{StatementWorthValue} = \Pi_i(\mathtt{StatementConjunct}_i\mathtt{WorthValue})$$
>
> The value of each conjunct is either the value of a constrained QoS characteristic or the value of a disjunction of constrained QoS characteristics. In the latter case, the worth value of the disjunction is the maximum worth value of its disjuncts:
>
> $$\mathtt{StatementConjunct}_i\mathtt{WorthValue} = \mathtt{MAX}_j(\mathtt{Characteristic}_j\mathtt{WorthValue})$$
>
> In most cases, the QoS statement only contains a list of constrained QoS characteristics. It is therefore in conjunctive normal form with each conjunct being a constrained QoS characteristic. The worth value of the QoS statement is then:
>
> $$\mathtt{StatementWorthValue} = \Pi_i(\mathtt{Characteristic}_i\mathtt{WorthValue})$$
>
> If one characteristic is more important than another is, it can be given a weighting function so that its worth function produces higher worth values, thereby strengthening its impact on the worth of the QoS statement. The default weighting function is a constant function with value 1 regardless of the value of the characteristic. Hence, QoS characteristics that do not have a specific weighting function do not impact the relative worth value of the QoS statement. The weighting function and the worth function can be combined into a single function that produces the weighted worth, in the following termed worth function.

Note that if a worth function evaluates to null for some values, the overall worth value will be null irrespective of the worth values of the other QoS characteristics.

The worth value of a QoS offer specified in conjunctive normal form is the product of the values of its conjuncts.

$$\texttt{OfferWorthValue} = \Pi_i(\texttt{OfferConjunct}_i\texttt{WorthValue})$$

The value of each conjunct is either the value of a statement or the value of a disjunction of statements. The value of a disjunction of statements is the maximum value of its disjuncts.

$$\texttt{OfferConjunct}_i\texttt{WorthValue} = \texttt{MAX}_j(\texttt{OfferDisjunct}_j\texttt{WorthValue})$$

In a QoS offer, each disjunctive statement contains only QoS statements as disjuncts.

$$\texttt{OfferDisjunct}_i\texttt{WorthValue} = \texttt{StatementWorthValue}$$

We consider definition of worth profiles to be outside the scope of CQML. The scheme outlined above represents one possible worth profile when individual worth functions are defined for the appropriate QoS characteristics. Other worth profiles may be designed that for instance allows dependencies between statement worth values (i.e., not only the product of the conjuncts) to calculate the overall worth value of the QoS offer. Nevertheless, the client can provide an interface that offers a worth value for QoS offers, calculated for instance following the scheme outlined above. We define an interface `WorthCalc` as:

```
interface WorthCalc {
  long worth(offer : Profile);
}
```

To enable the server to get the worth value of an offer, the client implements this interface:

```
component videoConsumer {
  uses VideoStream incoming;
  provides WorthCalc worth;
};
```

Any server may then use this interface to get the worth of its offer.

The approach of identifying a set of possible QoS offers can be combined with the worth-based approach. If so, it is preferable that for any client, the worth values of the predefined QoS offers will increase following the explicit order of the QoS offers.

QoS profiles and worth values are one-sided. However, in an executing system, agreements are made between collaborating parties where specific offers are chosen. Therefore we do not need a specific construct in CQML to represent a QoS contract; it is just a (set of) chosen profile(s) that have a limited period of validity. Agreements (including their QoS contracts) are either time-limited or event-limited, but we consider it a concern of the run-time system to represent such agreements.

4.6.5.4 Composition

In subsubsection 4.6.3.5, we defined the values of the QoS characteristics of composites. QoS profiles promise that a component will keep these values within certain ranges if the environment provides the appropriate conditions. We therefore need to consider composition of QoS profiles since, in the end, they reflect the QoS properties of the components that are composed. In QoS profiles, assumptions about the environment are made (QoS expectations). If we compose a set of components in which some parts of the composition make assumptions that are to be met by other parts of the composition (i.e., parts of the composition are dependent on each other), the QoS profile for the composition is not straightforward to relate to the QoS profiles for the individual parts. However, Abadi and Lamport provide profound insight into composition of specifications [Abadi and Lamport, 1990, Abadi and Lamport, 1993], and their results have been applied to QoS specifications in [Stefani, 1993, Leboucher and Najm, 1997], which is reflected in the work on QoS in ODP. Since a QoS profile in CQML is a way of specifying a QoS relation, the composition theorem applies if the QoS statements specify safety properties, i.e., properties that can be refuted in finite time (see section 4.7.5 for more details). Most QoS statements specify safety properties, although one can in CQML specify statements such as "maximum > x" that specify liveness properties not suited for composition. However, such properties can be converted to safety properties by introducing time limitations. This then means that most QoS profiles for compositions can be derived based on QoS profiles for its parts. The process of performing such derivations is, however, outside the scope of CQML.

4.6.5.5 Keywords

To define QoS profiles, we have introduced the following keyword:

profile | **for** | **uses** | **provides** | **worth** | **transition** | **any** | **precedence**

4.6.6 QoS Categories

QoS characteristics, QoS statements and QoS profiles can be grouped into QoS categories. Different application domains have different sets of relevant QoS terminology, and we group such terminology into QoS categories. A QoS category represents a naming context and it may be nested. The rationale for grouping is that particular systems provide specific categories of QoS characteristics, QoS statements and QoS profiles, and we want to be able to reflect this. For instance, a different set of QoS aspects is relevant for a multimedia application than for an online banking application, although both provide a level of QoS to its users. In addition, by introducing categories we resolve naming conflicts. For instance, the apparently same characteristic such as "availability" may in a security context mean possibility of malicious intervention while it may in the context of service execution mean failure rate. QoS categories have the following syntax:

```
<category_declaration> ::= quality_category <category_name>
                           '{' <category_body> '}'
<category_body>        ::= <cqml_declaration>+
                       |   <cqml_reference>+
<cqml_reference>       ::= <characteristic_name> ';'
                       |   <quality_name> ';'
                       |   <profile_name> ';'
                       |   <category_name> ';'
```

We can, for instance, group the QoS characteristics `output` and `delay` into the `timeliness` category:

```
quality_category timeliness {
      output;
      delay;
}
```

4.6.6.1 Keywords

To define QoS categories, we have introduced the following keyword:

quality_category

4.6.7 Names and Scoping

The name and scoping rules of CQML are based on those of CORBA IDL. The objective of CORBA is to provide a platform-independent infrastructure for distributed computing, and the name and scoping rules are therefore designed to provide maximum interoperability with other languages. Since CQML shares the objective of platform-independence and CORBA IDL represents one important interface definition language that CQML must interoperate with, it is the natural choice to base its name and scoping rules on this.

CQML uses the same rules for case-sensitivity as CORBA IDL. The definition of a type that only differs from another type in the case of its characters is considered a redefinition. However, to allow natural mappings to infrastructures implemented in case-sensitive languages, all references to a type must use the same case as in the definition.

CQML defines hierarchical namespaces using nested QoS categories. If no QoS category is specified, definitions using CQML are in the default QoS category `QUALITY`. A QoS category is parallel to a CORBA module. The default QoS category is provided to avoid name clashes with other parts of the system.

In addition to QoS categories, QoS profiles define namespaces that may encapsulate simple QoS profiles. A simple QoS profile is a QoS profile that is part of the definition of another QoS profile, and cannot contain QoS profiles itself. Enumerations and sets defined in a QoS characteristic also define namespaces for the user-defined names they contain. Finally, in OCL expressions, a namespace may be created using the OCL-keyword `let`.

The following identifiers are defined in CQML: QoS categories, QoS profiles, QoS statements, QoS characteristics, attributes, enumeration values and set values. In contrast to CORBA IDL, the definition of some identifiers may be overloaded. The definition of the following identifiers may be overloaded: QoS profiles, QoS statements, and QoS characteristics.

Identifiers can be used in its namespace or any namespaces within it. An identifier can also be used across QoS categories by referencing the QoS category using dot ('.') as the scope resolution operator. This is similar to referencing in Java packages.

In contrast to CORBA IDL, a CQML file does not form a namespace. To avoid compilation errors, a CQML file must be complete in the sense that all definitions must end in the file; open parentheses will give compilation errors. QoS categories can, however, be reopened in another file and additional definitions in the QoS category can be given. To make previous definitions available for referencing, the compiler needs access to the namespace hierarchy previously defined, not only the namespace hierarchy that is defined in the file that is being compiled. This is also similar to the approach in Java where references outside the file causes the proper `.class` file to be loaded.

4.6.8 Summary and Evaluation

4.6.8.1 Overview

In Figure 19, a model in UML of the concepts in CQML is shown. The classes with name in bold correspond directly to concepts in CQML, while the relationships with name in bold correspond to keywords in CQML. This model is refined into a UML profile for QoS in section 5.4.

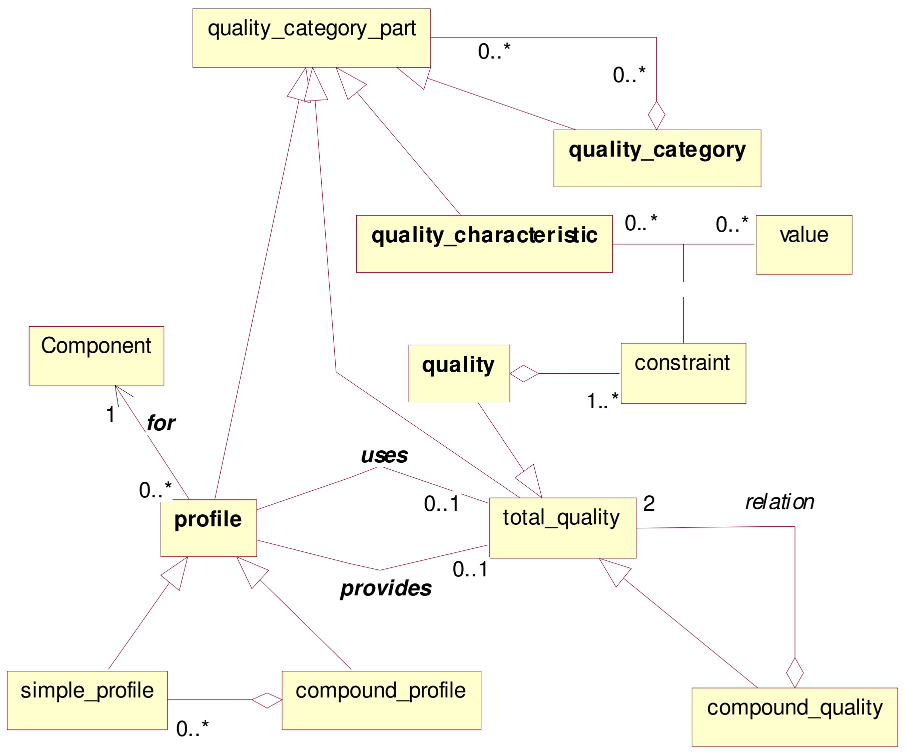


Figure 19: CQML meta-model

4.6.8.2 Evaluation

Here we position CQML in the evaluation framework used to evaluate existing approaches in subsection 4.4.9, where each requirement from section 4.3 is listed and it is indicated to what extent each approach supports it. We have replaced formal methods with CQML in the table, but have repeated the rest of the table for convenience. How CQML addresses each requirement is discussed after the table.

Requirement	CQML	SM-IL	TINA ODL	MAQS QIDL	QuO	QML	QDL
1. Generality	++	--	+	+	+	++	+
2. Life-cycle	++	--	+	+	+	++	+
3. IDL integration	++	--	+/-	+/-	+	++	?
4. Platform-independence	++	--	++	++	++	++	++
5. RM-ODP compliance	++	--	++	+	+	+	+
6. Object-oriented	++	+	++	++	+	++	++
7. UML integration	+	--	-	-	-	+	-
8. Separation	++	+	--	--	-	++	++
9. Syntactic separation	++	--	--	--	++	++	?
10. Predefined adaptation	++	+	--	--	++	--	-
11. Negotiation	+	--	--	--	--	+	--
12. Adaptive behaviour	++	--	--	--	++	--	--

13. Composition	++	+	--	--	--	--	+
14. Refinement	++	--	--	++	++	++	++
15. Typed	++	+	-	++	++	++	++
16. Precision	++	--	--	--	--	--	--
17. Interface annotation	++	--	--	++	++	++	?
18. Component annotation	++	--	--	--	++	++	?
19. Combinations	++	--	--	--	--	--	?
20. Expectations and offers	++	--	--	--	+	+	++
21. Code generation	+	--	+	++	++	+	+
22. Agreement process	++	--	-	-	-	++	?
23. Adaptation	++	+	--	-	++	+	-
24. Observable	++	++	?	+	++	++	?
25. Comparison	++	--	-	+	+	+	?

Table 3: Evaluation table including CQML

From the table we see that CQML satisfies the requirements identified in section 4.3. Compared with QML, the approach that in section 4.4 were evaluated to satisfy most of the requirements, CQML adds support for precise specification of QoS and support for predefined adaptation and adaptive behaviour. In addition, CQML includes support for composition of QoS specifications. Below we discuss how CQML addresses each of the requirements from section 4.3:

1. Generality – In CQML, any type of QoS characteristic can be specified. The definition of QoS characteristics uses concepts in application models from any domain using the general-purpose constraint language OCL.
2. Life-cycle support – CQML does not restrict when in the software development process the specifications are used. On the contrary, using CQML it is possible to specify both subjective, user-oriented QoS characteristics and objective, system-oriented QoS characteristics. In particular, QoS characteristics can be refined supporting the top-down approach of software development. Also, existing QoS characteristics can be used when specifying profiles, supporting the bottom-up approach to software development.
3. Integration with well-known IDLs – CQML can be used with any interface definition language since specifications in CQML are separated from those in IDL.
4. Platform-independence – CQML does not use features of any underlying platform, it uses an abstract computational model that can be supported by many platforms.
5. RM-ODP compliance – CQML is based on a computational model compliant with RM-ODP.
6. Object-oriented – CQML uses an object-oriented computational model with objects and interfaces as basic modelling concepts.
7. UML-integration – CQML uses OCL, an integral part of UML, to specify the values of QoS characteristics. Furthermore, the concepts of CQML are integrated with the UML meta-model as a UML profile in section 5.4.

8. Separation of functional and non-functional aspects in component specifications – CQML focuses on non-functional aspects and only references the functional aspects in the component specification.
9. Syntactic separation – Specifications in CQML are syntactically separate from interface definitions.
10. Predefined adaptation – CQML supports predefined adaptation by nested profiles and precedences between them.
11. Negotiation – CQML supports negotiation by providing facilities for specifying the existence of worth functions that service providers can use.
12. Adaptive behaviour – CQML supports the triggering of behaviour when predefined adaptation occurs by making it possible to specify profile transitions that trigger operations. These operations can either be operations in the outgoing or incoming interfaces of the component, i.e., on the client or server, respectively.
13. Composition – CQML supports specification of QoS characteristic values of compositions by the three keywords **`parallel-and`**, **`parallel-or`** and **`sequential`**. Furthermore, resulting QoS profiles of compositions can in many cases be derived using the Abadi-Lamport composition theorem.
14. Refinement – CQML supports specialisation of **`quality_characteristic`**, **`quality`** and **`profile`**.
15. Typed – **`quality_characteristic`**, **`quality`** and **`profile`** all define types.
16. Precision – CQML supports precise specification by invariants and the **`values`** clause of QoS characteristics.
17. Interface annotation – Profiles in CQML support this.
18. Component annotation – This is supported in CQML by profiles that associate QoS statements with component specifications.
19. Different combinations – CQML supports different combinations of components with QoS properties by the three keywords **`parallel-and`**, **`parallel-or`** and **`sequential`**. Also, the **`uses`**-clause denotes dependencies between components.
20. Expectations and offers – CQML supports specification of both what a client needs of QoS and what the service provides using the **`uses`** and **`provides`** keywords.
21. Code generation – CQML supports code generation by being well-defined and supporting precise definitions of the QoS characteristics as requirement 16 addresses. Precise definitions help the code generator in producing appropriate code for QoS support. However, no code generator has been implemented.
22. Dynamic and static agreement process – CQML supports specification of static QoS contracts by using QoS profiles that denote a specific level of QoS. Both dynamic matching of statically defined profiles and dynamic specification of new profiles to meet current requirements can be implemented by the infrastructure since profiles are first-class objects that can be inspected and manipulated at run-time.
23. Re-negotiation support– CQML supports specification of several QoS profiles, thereby facilitating application adaptation using re-negotiation supported by the infrastructure. Application adaptation is also supported by the transitional behaviour that can be specified since it may trigger appropriate operations in the application.
24. Observable – CQML supports observable QoS characteristics by providing facilities to specify the semantics of QoS characteristics in the **`values`**-clause. Such specifications may help the infrastructure when monitoring and reporting the results.

25. Comparison of values of distributed components – CQML supports distribution transparency and comparison between values are made in CQML without distribution concerns. However, the underlying event model includes event contexts that may include location information that the supporting infrastructure can use to compare values from different locations.

4.7 Semantics

It is useful to have a clear understanding of conformance between specifications for two reasons. Firstly, when one wants to use one component in place of another (substitutability), conformance between the two component specifications is required. Secondly, when one tries to match specifications in order to establish a contract, the offered properties of one must conform to some of the required properties of the other. In either case, both functional and non-functional aspects of the component's specifications must be considered. In this thesis, however, only the non-functional aspects are discussed. For functional conformance we rely on results such as those reported in [Najm and Stefani, 1995] for the ODP computational model, and [Eliassen and Mehus, 1998, Eliassen and Rafaelsen, 1999] for type checking of streams. In this section, we discuss what is meant by conformance between two QoS profiles, both in the context of substitutability and in the context of partial conformance, in order to be able to establish QoS contracts.

Several actors care about conformance. The QoS framework needs an understanding of conformance when performing QoS management. For instance, a QoS framework may run background conformance tests over registered QoS profiles and build indexes over substitutable components. These indexes can later be used to establish QoS contracts among components or to replace a component with another as part of adaptation. A QoS framework may also include a validity checker that evaluates whether the QoS expectations in a QoS profile can be met so that it can be marked as valid. When evaluating QoS expectations, the validity checker needs to find valid QoS offers that conform to the QoS expectations. The validity checker can be used to select matching components and establish QoS contracts between them. A component developer may also use such a validity checker when deploying the component to check whether it is usable in the context in which it is deployed or if not, discover what QoS expectations that cannot be met.

In order to discuss conformance, however, we first define a formal representation of the three kinds of constructs in CQML that have semantic impact.

4.7.1 Formal Representation

We use the following notation. A value is specified in *italic*. A set is specified in Arial font. <> enclose the elements in a tuple so that $<a, b>$ denotes the pair (2-tuple) with a and b as elements, while { } enclose the elements of a set so that $\{a, b\}$ denotes the set with a and b as members. Regular parentheses () are used for grouping. $\mathbf{2}^{\mathsf{Set}}$ denotes the power set of Set (i.e., the set of all sets that contain elements from Set), while K_{Tuple} is used for the index set of Tuple (i.e., for a tuple $<x_1, \ldots, x_n>$, K_{Tuple} would be $\{1, \ldots, n\}$). A – B denotes set difference, i.e., the set of elements that are members of A but not members of B. #A represents the cardinality of set A.

4.7.1.1 QoS Characteristic

The following tuple defines a QoS characteristic *qoschar*:

$$qoschar = <name, qosCharDomain, pred, <_{result}, formalParams, valueSpec, stat, comp> \in \mathsf{QoSChar}$$

where

$name \in \mathsf{Name}$,

$qosCharDomain \in \mathsf{QoSCharDomain} = \{\mathsf{Domain}_1, \mathsf{Domain}_2, \mathsf{Domain}_3\}$,
where $\mathsf{Domain}_1 \subseteq \mathbb{R}$, $\mathsf{Domain}_2 \subseteq \mathsf{Name}$ and $\mathsf{Domain}_3 \subseteq \mathbf{2}^{\mathsf{Name}}$,

$pred \in \mathsf{Pred}$,
where Pred is the set of all predicates,

$<_{result} \in \mathsf{Order} = \mathsf{OrderRelation} \times \{increasing, decreasing\}$,
where OrderRelation is an ordering of *qosCharDomain*,

$formalParams = <param_1, \ldots, param_n> \in \mathsf{FormalParams} = \prod_{i=1}^{n} (\mathsf{Name} \times \mathsf{Any})$,
where n is the number of parameters and Any denotes the set of all types,
$param = <name, type> \in \mathsf{Name} \times \mathsf{Any}$,

$valueSpec \in \mathsf{ValueSpec} = (\mathsf{FormalParams} \rightarrow \mathsf{QoSCharDomain}) \cup \{Undefined\}$,

$stat \subseteq \mathsf{Stat} = \{maximum\} \cup \{minimum\} \cup \{range\} \cup \{mean\} \cup \{variance\} \cup \{standard\_deviation\} \cup (\{percentile\} \times \mathsf{Number}_1) \cup (\{moment\} \times \mathsf{Number}_2) \cup (\{frequency\} \times \mathsf{Range}) \cup (\{distribution\} \times \mathsf{Name})$,
where Number_1 is the set of integers [0, …, 100], Number_2 is the set of non-negative integers, and Range is the set of intervals of *qosCharDomain* under the order relation in $<_{result}$ (i.e., the possible value ranges),

$comp \in \mathsf{Comp} = \mathsf{CompFunc}_{parallel\text{-}or} \times \mathsf{CompFunc}_{parallel\text{-}and} \times \mathsf{CompFunc}_{sequential}$,
where $\mathsf{CompFunc}_{parallel\text{-}or}$, $\mathsf{CompFunc}_{parallel\text{-}and}$, $\mathsf{CompFunc}_{sequential}$ are different composition functions for the three types of compositions and CompFunc is a projection of two QoS characteristics onto a new QoS characteristic, all of the same type: $\mathsf{CompFunc} = (\mathsf{QoSCharDomain} \times \mathsf{QoSCharDomain} \rightarrow \mathsf{QoSCharDomain}) \cup \{Undefined\}$.

A QoS characteristic has a name termed *name* that is a member of the set of names denoted Name.

A QoS characteristic has a domain termed *qosCharDomain* that is a member of QoSCharDomain. This domain is a subset of the numeric, enum or set domains. From the basic definition of a QoS characteristic as numeric, enum or set, it may be further restricted in a number of ways. For the numeric domain, it may be restricted to be only the set of integers or the set of natural numbers (including zero), and a value range may be specified that also restricts it. For enum and set, the set of names must be restricted to a finite set of names, e.g., by listing every member in the definition of the QoS characteristic. Finally, an invariant may be specified that further constrains the value range. If we denote the base domain of a QoS characteristic (i.e., any one of numeric, enum or set) QoSCharDomainBase,

$$\mathsf{QoSCharDomainBase} \in \{\mathbb{R}\} \cup \{\mathsf{Name}\} \cup \{\mathbf{2}^{\mathsf{Name}}\},$$

these restrictions may collectively be defined as a membership predicate *pred* on the QoSCharDomainBase so that

$$\mathsf{QoSCharDomain} = \{x \in \mathsf{QoSCharDomainBase} \mid pred(x)\},$$

i.e., QoSCharDomain is the result of applying the membership predicate to the base domain.

A QoS characteristic has a resulting order of its domain $<_{result}$ that is a member of Order. The order may be partial, total or undefined. $<_{result}$ is defined based on a value order. For the numeric domain, the common pre-defined total ordering of numbers is used (denoted '<') as the value order. For the enum domain, a value order can be defined as a list of ordered pairs of the user-defined names. Such an ordering is transitive. The set domain is always at least partially ordered. The definition of the value order of the set domain depends on whether the user-defined names that are used in the power set are ordered or not. If they are not ordered, the value order of the set domain corresponds to set inclusion and the value order is partial. If they are ordered, a user-defined name in a set A does not have to be included in a set B if it is ordered below at least one of the user-defined names in B that is not in A:

$$\begin{aligned} \mathsf{A} <_{\text{val}} \mathsf{B} &\Leftrightarrow \forall a \in (\mathsf{A} - \mathsf{B}), \exists b \in (\mathsf{B} - \mathsf{A}) \mid a <_{\text{elem}} b \\ &\Leftrightarrow \max(\mathsf{A} - \mathsf{B}) <_{\text{elem}} \max(\mathsf{B} - \mathsf{A}) \end{aligned}$$

where $<_{\text{val}}$ represents the ordering relation between the two values (here: elements in the power set) and $<_{\text{elem}}$ represents the ordering of individual user-defined names and $\max(\emptyset)$ is defined as a value less than any other value (often denoted $\perp$). For example, if the set of names is {a,b,c} and these are ordered (totally) as a $<_{\text{elem}}$ b $<_{\text{elem}}$ c, the (total) value order of the power set is: $\emptyset <_{\text{val}} \{a\} <_{\text{val}} \{b\} <_{\text{val}} \{a,b\} <_{\text{val}} \{c\} <_{\text{val}} \{a,c\} <_{\text{val}} \{b,c\} <_{\text{val}} \{a,b,c\}$.

As part of Order, it can be specified whether increasing values correspond to increasing or decreasing semantic worth. If the domain is defined as increasing, the value ordering is preserved in the resulting order, while if it is defined as decreasing, it is reversed. With $<_{\text{result}}$ and $<_{\text{val}}$ already defined, the following applies:

Increasing: $\forall x,y \in \mathsf{QosCharDomain} \mid (x <_{\text{val}} y) \Rightarrow (x <_{\text{result}} y)$
Decreasing: $\forall x,y \in \mathsf{QosCharDomain} \mid (x <_{\text{val}} y) \Rightarrow (y <_{\text{result}} x)$

A QoS characteristic has a (possibly empty) list of formal parameters *formalParams* from a set of formal parameters termed FormalParams. The elements in FormalParams are named types and can be any type (object type, operation type, stream type, or signal type).

A QoS characteristic has a definition of how properties of the parameters can be combined to give the value of the QoS characteristic when actual parameters are provided, for instance at run-time in a QoS framework. The definition *valueSpec* is a member of the set of value specifications termed ValueSpec and is either undefined or one of the projections from the value space (type) of the actual parameters provided for the FormalParams onto QoSCharDomain, i.e., a specification of how one can derive the value of the characteristic based on the values of the formal parameters.

Stat is the set of statistical aspects defined for the QoS characteristic. *stat* is a subset of the statistical aspects possible to define. Range is the set of value ranges:

$$\begin{aligned} \mathsf{Range} = \{ \langle &lowerBound, lowerValue, upperValue, upperBound\rangle \mid \\ &lowerBound, upperBound \in \{open, closed\}, \\ &lowerValue, upperValue \in qosCharDomain \cup \{Undefined\}, \\ &lowerValue \neq Undefined \vee upperValue \neq Undefined \} \end{aligned}$$

Comp is the set of possible composition functions for each of the three composition cases. A composition *comp* has a composition function (that may be undefined) for each of the three types of compositions. For instance, *comp* may specify regular addition (+) for sequential composition of a QoS characteristic with a numeric domain and be undefined for the two other cases.

4.7.1.2 QoS Statement

A QoS statement *qosStat* has a name, an *n*-tuple of constrained and qualified QoS characteristics, a list of formal parameters, and a list of bindings of these formal parameters to the parameters of the QoS characteristics:

$$qosStat \in \mathsf{QoSStat} = \mathsf{Name} \times \mathsf{ConstrQoSChars} \times \mathsf{QoSStatFormalParams} \times \mathsf{QoSStatBindings}$$
$$constrQoSChars \in \mathsf{ConstrQoSChars} = \mathsf{ConstrQoSChar}_1 \times \ldots \times \mathsf{ConstrQoSChar}_n$$

where *n* is the number of QoS characteristics used in this QoS statement.

ConstrQoSChar is defined by a QoS characteristic (QoSChar) and a discriminator Discriminator:

$$constrQoSChar = \langle qosChar, discriminator\rangle \in \mathsf{ConstrQoSChar} = \mathsf{QoSChar} \times \mathsf{Discriminator}$$
$$discriminator \in \mathsf{Discriminator} = \{guaranteed\} \cup \{best\text{-}effort\} \cup (\{best\text{-}effort\} \times \mathsf{Qualifier})$$
$$qualifier \in \mathsf{Qualifier} = \{threshold, compulsory\} \times \mathsf{QoSChar.QosCharDomain}$$

where an instance of QoSChar.QoSCharDomain in Qualifier represents the limit value for the qualification.

The value ranges of the QoS characteristics are further restricted from their definitions by a constraining predicate *QoSStatPred* so that for the domains *restrictedDomain*$_1$,..., *restrictedDomain*$_n$ of the n QoS characteristics, the following apply:

$$\langle restrictedDomain_1,\ldots, restrictedDomain_n\rangle = \langle x_1,\ldots, x_n\rangle \mid$$
$$(\, (\forall i \in \{1..n\} \mid x_i \subseteq \mathsf{ConstrQoSChar}_i\mathsf{.QoSChar.QoSCharDomain}) \wedge$$
$$(\forall val_1, \ldots, val_n \in x_1,\ldots, x_n \mid QoSStatPred(val_1, \ldots, val_n))\,)$$

A QoS statement is a conjunction of constraints where each conjunct may be an arbitrarily complex logical expression of constrained QoS characteristics. However, since a QoS statement can be expressed in its conjunctive normal form where each conjunct is a disjunction of constraints for one QoS characteristic, the maxset normal form representation[6] above is sufficient.

The formal parameters *qosStatFormalParams* of the QoS statement are defined as an *f*-tuple of named types:

$$qosStatFormalParams = \langle fp_1, \ldots, fp_f\rangle \in \mathsf{QoSStatFormalParams} = \prod_{i=1}^{f} (\mathsf{Name} \times \mathsf{Any})$$
$$fp = \langle name, type\rangle \in \mathsf{Name} \times \mathsf{Any}$$

where *f* is the number of parameters. A QoS statement cannot have more formal parameters than the number of formal parameters for its constrained QoS characteristics. Hence, with *n* constrained QoS characteristics, the following applies:

$$\mathrm{f} <= \sum_{i=1}^{\mathrm{n}} (\# K_{\mathsf{ConstrQoSChar}_i\mathsf{.QoSChar.FormalParams}})$$

where $\# K_{\mathsf{ConstrQoSChar}_i\mathsf{.QoSChar.FormalParams}}$ denotes the number of formal parameters (the cardinality of the index set) for QoS characteristic *i*.

In a QoS statement, the formal parameters are used as actual parameters in the QoS characteristics, so there exists a binding of each formal parameter in all *n* QoS characteristics with a formal parameter of the QoS statement:

$$qosStatBindings = \langle singleStatBinding_1, \ldots, singleStatBinding_n\rangle \in \mathsf{QoSStatBindings}$$

$$\mathsf{QoSStatBindings} = (\prod_{i=1}^{n} \prod_{j=1}^{\#K_{\mathsf{ConstrQoSChars}_i.qosChar.formalParams}} \mathsf{ConstrQoSChars}_i.qosChar.formalParams_j \rightarrow \mathsf{QoSStatFormalParams})$$

where an individual binding is defined as:

[6] The maxset normal form corresponds to the conjunctive normal form of membership predicates.

$$\mathit{singleStatBinding} = \langle \mathit{fpchar}, \mathit{fpstat}\rangle \mid ($$

$$\mathit{fpchar} \in (\bigcup_{i,j} \{\mathsf{ConstrQoSChars}_i.\mathit{qosChar.formalParams}_j \mid (i \in K_{\mathsf{ConstrQoSChars}}, j \in K_{\mathsf{ConstrQoSChars}_i.\mathit{qosChar.formalParams}})\}) \wedge$$

$$\mathit{fpstat} \in \bigcup_{i \in K_{\mathsf{QoSStatFormalParams}}} \{\mathsf{QoSStatFormalParams}_i.\mathit{type}\}$$

In this definition, *fpchar* denotes the formal parameter for the QoS characteristic while *fpstat* denotes the formal parameter of the QoS statement.

Note that the type of a formal parameter in a QoS statement may be renamed or subtyped from its representation in the QoS characteristic. However, we exclude that in this model. It provides no additional insights and can be dealt with in a straightforward manner.

4.7.1.3 QoS Profile

A QoS profile is defined for a particular component type for which it is a profile. A QoS profile is defined as:

$$\mathit{qosProfile} = \langle \mathit{componentType}, \mathit{profileBody}\rangle \in \mathsf{QoSProfile} = \mathsf{Any} \times \mathsf{ProfileBody}$$
$$\mathit{profileBody} \in \mathsf{ProfileBody} = \mathsf{SimpleProfile} \cup \mathsf{CompoundProfile}$$

where *componentType* is the type of the component for which it is a profile, and ProfileBody is either a simple or compound profile. A simple profile contains a uses- and a provides-clause and an invariant:

$$\mathit{simpleProfile} = \langle \mathit{uses}, \mathit{provides}, \mathit{invariant}\rangle \in \mathsf{SimpleProfile} = \mathsf{Uses} \times \mathsf{Provides} \times \mathsf{Pred}$$
$$\mathit{uses} \in \mathsf{Uses} = \mathsf{Conjunct}_1 \times \ldots \times \mathsf{Conjunct}_m$$
$$\mathit{provides} \in \mathsf{Provides} = \mathsf{Conjunct}_1 \times \ldots \times \mathsf{Conjunct}_n$$
$$\mathit{invariant} \in \mathsf{Pred}$$

where Pred is the set of all predicates and where both clauses are in the conjunctive normal form (maxset normal form), and m and n are the number of conjuncts in the uses- and provides clause, respectively. Note that a clause in conjunctive normal form is a conjunction, and each term in a conjunction is called a conjunct. Similarly, a disjunction consists of a number of disjuncts.

In conjunctive normal form, a conjunct is a disjunction of p bound QoS statements:

$$\mathsf{Conjunct} = \mathsf{BoundQoSStat}_1 \times \ldots \times \mathsf{BoundQoSStat}_p$$

where p is the number of different QoS statements in the disjunction. Such a bound QoS statement is a QoS statement where each formal parameter in each QoS characteristic used in any part of the QoS statement is bound by elements of the component type:

$$\mathsf{BoundQoSStat} = \mathsf{QoSStat} \times \mathsf{ComponentBindings}$$
$$\mathsf{ComponentBindings} = \mathsf{QoSStat.QoSStatFormalParams} \to \mathit{componentType}$$
$$= \{ \langle \mathit{singleCompBinding}_1, \ldots, \mathit{singleCompBinding}_n\rangle \mid$$
$$\mathit{singleCompBinding} = \langle \mathit{fp}, \mathit{ce}\rangle \mid$$
$$(\mathit{fp} \in \bigcup_i \{\mathsf{QoSStat.QoSStatFormalParams}_i \mid \mathrm{i} \in K_{\mathsf{QoSStat.QoSStatFormalParams}}\},$$
$$\mathit{ce} \in \mathsf{Type}_x \text{ in } \mathit{componentType})$$

where Type_x in *componentType* denotes that an element of type Type_x is part of *componentType*.

A compound profile consists of a number of named profiles, transitions between such named profiles and an ordering (precedence) of them:

$$\mathit{compoundProfile} \in \mathsf{CompoundProfile} = \mathsf{NamedProfiles} \times \mathsf{Transitions} \times \mathsf{Precedence}$$

The named profiles are simple profiles with names:

$$\begin{aligned} namedProfiles \in \text{NamedProfiles} = \{\, & \langle namedProfile_1, \ldots, namedProfile_k \rangle \mid \\ & namedProfile = \langle profileName, simpleProfile \rangle \mid \\ & (\, profileName \in \text{Name}, \\ & simpleProfile \in \text{SimpleProfile}\,)\,\} \end{aligned}$$

where k is the number of named profiles. For a compound profile with l transitions, the transitions are:

$$\begin{aligned} \text{Transitions} = \{ & \langle transition_1, \ldots, transition_l \rangle \mid \\ & transition = \langle from, to, ops \rangle \mid \\ & \quad from \in \text{From} = \text{Name} \cup \{any\}, \\ & \quad to \in \text{To} = \text{Name} \cup \{any\}, \\ & \quad ops = \langle op_1, \ldots, op_m \rangle \mid \\ & \quad\quad op \in \text{ComponentType.Operation}\ \} \end{aligned}$$

where m is the number of operations in a transition and ComponentType.Operation is the set of operations that is defined for the component.

Precedence is a total order of the names of the k profiles:

$$precedence = \langle name_1, \ldots, name_k \rangle \in \text{Precedence} = P_k(\text{ProfileName})$$

where $name_i \in$ ProfileName and P_k(ProfileName) is the set of all permutations of ProfileName. ProfileName is the set of profile names used in the compound profile that contains the precedence. Note that the precedence specifies the order of selection of simple profiles within a compound profile. However, a number of different profiles (simple or compound) may have been specified for a component, but the precedence does not consider transitions to such other profiles for the component.

4.7.2 Substitutability

In this subsection, we discuss the properties that define when a component with a certain QoS profile is substitutable for a component with another QoS profile. Substitutability is especially important when performing service initialisation and adaptation. In order to be able to pick appropriate components at service initialisation, a component broker needs to know the set of components that has QoS offers and QoS expectations that conform to some given requirements, i.e., that are substitutable with a virtual component that exactly fits the requirements. When performing adaptation, the QoS framework may need to substitute a component with another that are better suited to work in the new environment and a list of substitutable components would be useful for this purpose.

4.7.2.1 QoS Profiles

A QoS-enabled component specification consists of a component specification and a set of QoS profile specifications (simple or compound). A QoS profile for a component specifies a type that depends on the component type, so conformance between two QoS profiles depends on conformance between the component types. Hence, a QoS profile A conforms to a QoS profile B (denoted $A \propto_{\text{profile}} B$), iff the component type in A conforms to the component type in B and the profile body in A conforms to the profile body in B:

$$\begin{aligned} A \propto_{\text{profile}} B \Leftrightarrow & (\text{ComponentType}_A \propto_{\text{type}} \text{ComponentType}_B) \wedge \\ & (\text{ProfileBody}_A \propto_{\text{profileBody}} \text{ProfileBody}_B) \end{aligned}$$

If $A \propto_{\text{profile}} B$, a component with a QoS profile A can be used wherever a component with the profile B can be used, i.e., conformance implies substitutability of the component.

Component types are defined in a component definition language outside CQML. We assume the necessary conformance rules for such types are defined together with this component definition language, and we do not discuss component type conformance $\propto_{\text{type}}$ further here. Moreover, we are only concerned in this context about conformance of those parts of the component specification that are used in the profiles.

We need, however, to further elaborate on conformance between the profile bodies. A profile body is either a simple or a compound profile; we need to define conformance in both cases:

$$\begin{aligned} C \propto_{\text{profileBody}} D \Leftrightarrow \quad & (C \in \mathsf{SimpleProfile} \Rightarrow C \propto_{\text{simpleProfile}} D) \wedge \\ & (C \in \mathsf{CompoundProfile} \Rightarrow C \propto_{\text{compoundProfile}} D) \end{aligned}$$

where C and D are two profile bodies. We need two different conformance relations since $\propto_{\text{compoundProfile}}$ needs to address conformance of the transitions and precedence ordering in addition to the issues that $\propto_{\text{simpleProfile}}$ addresses. Note that the specification above does not restrict the profile type of D, we address both profile types for both conformance relations.

4.7.2.1.1 Two Simple Profiles

If C and D are simple profiles, C is conformant to D iff it provides at least the same and expects at most the same properties as D, and the invariant of D is valid whenever the invariant of C is. In other words, the uses- and provides clauses in C must conform to the ones in D, and the invariant in C must imply the one in D:

$$\begin{aligned} C \propto_{\text{simpleProfile}} D \Leftarrow \quad & (C \in \mathsf{SimpleProfile}) \wedge (D \in \mathsf{SimpleProfile}) \wedge \\ & (C.provides \propto_{\text{provides}} D.provides) \wedge (C.uses \propto_{\text{uses}} D.uses) \wedge \\ & (C.invariant \Rightarrow D.invariant) \end{aligned}$$

The properties a component provides according to a QoS profile are specified in the provides-clause. If the provides-clauses in C and D are denoted E and F, respectively, E conforms to F if every property in F is also a property in E. Formally, E conforms to F iff each conjunct of disjunctive QoS statements in F has a conjunct of disjunctive QoS statements in E that conforms to it:

$$E \propto_{\text{provides}} F \Leftrightarrow \forall j \in \mathsf{Conjunct}_F, \exists i \in \mathsf{Conjunct}_E \mid i \propto_{\text{conjunct}} j$$

The properties a component needs in order to operate successfully are specified in the uses-clause. If the uses-clauses in C and D are denoted G and H, respectively, G conforms to H if G poses no further restrictions on the environment from what is specified in H. G may, however, relax the constraints from the ones in H. Formally, G conforms to H iff each conjunct of disjunctive QoS statements in G has a conjunct of disjunctive QoS statements in H that conforms to it:

$$G \propto_{\text{uses}} H \Leftrightarrow \forall j \in \mathsf{Conjunct}_G, \exists i \in \mathsf{Conjunct}_H \mid i \propto_{\text{conjunct}} j$$

Note the difference between the uses- and provides-clauses. The expectations of a component from its environment can at most be those of the component it conforms to, while the offers are at least the ones of the component it conforms to offers.

Both the definitions of conformance for the provides- and uses clauses rely on conformance between conjuncts of disjunctive QoS statements. Such a disjunction conforms to another iff each disjunct of the first conforms to a disjunct in the other. At any point in time, only one of the disjuncts have to conform to a disjunct in the other, but since we cannot determine which one a priori, conformance requires that all disjuncts conform to a disjunct in the corresponding disjunction. Formally, if we denote two disjunctions of QoS statements I and J, I conforms to J iff the QoS statements in I conforms to a QoS statement in J:

$$I \propto_{\text{conjunct}} J \Leftrightarrow \forall k \in \mathsf{BoundQoSStat}_I, \exists l \in \mathsf{BoundQoSStat}_J \mid k \propto_{\text{boundStat}} l$$

Note that the QoS statements in a QoS profile are bound, i.e., their formal parameters are bound to elements of the component type. We therefore need to have conformance of both the QoS statements and the bindings of their formal parameters. Denoting two bound QoS statements K and L, the conformance can be specified as:

$$K \propto_{\text{boundStat}} L \Leftrightarrow \; (K.\mathsf{QoSStat} \propto_{\text{statement}} L.\mathsf{QoSStat}) \wedge (K.\mathsf{ComponentBindings} \propto_{\text{compBindings}} L.\mathsf{ComponentBindings})$$

We postpone the discussion of conformance between QoS statements to subsubsection 4.7.2.2 when all aspects of profile conformance are defined.

On the binding conformance, we have already established that the component types must conform. Hence, all elements in the component type to which *L* applies must have elements that conform to them in the component type to which *K* applies. Also, we assume that the QoS statements in the two bound QoS statements conform, and hence, the formal parameters of the QoS statement in *K* conform to the formal parameters of the QoS statement in *L*. Having these facts in mind, the binding of the formal parameters in *K* conforms to the binding of the formal parameters in *L* iff the formal parameters in *K* that have corresponding formal parameters in *L*, are bound by elements in the component type of *K* that conform to elements in the component type of *L*. In other words, the formal parameters in *K* that conform to parameters in *L* must be bound by conforming elements of the component types. If we denote two bindings *M* and *N*, that *M* conforms to *N* is defined as:

$$M \propto_{\text{compBindings}} N \Leftrightarrow \forall sb_M \in M.\mathsf{SingleCompBinding}, \exists sb_N \in N.\mathsf{SingleCompBinding} \mid ((sb_M.fp \propto_{\text{qosStatFormalParam}} sb_N.fp) \wedge (sb_M.ce \propto_{\text{type}} sb_N.ce))$$

where $M.\mathsf{SingleCompBinding}$ denotes the set of single bindings in *M*:

$$M.\mathsf{SingleCompBinding} = \bigcup_{i \in K_{\mathsf{M.ComponentBindings}}} \{M.\mathsf{ComponentBindings}_i\}$$

Conformance between formal parameters $\propto_{\text{qosStatFormalParam}}$ will be defined when conformance between QoS statements is discussed, and, as previously discussed, conformance between component types $\propto_{\text{type}}$ is outside the scope of this work.

4.7.2.1.2 Simple Profile Conforms to Compound Profile

We have now defined conformance between QoS profiles *C* and *D* in the case that both are simple profiles. We also need to consider the case where *D* is compound. A simple profile *C* conforms to a compound profile *D* if it conforms to each simple profile that is part of *D*. Further, since a simple profile does not contain any transitional behaviour, *D* cannot contain any transitional behaviour if *C* is to conform to it. Formally, it is specified as:

$$C \propto_{\text{simpleProfile}} D \Leftarrow (C \in \mathsf{SimpleProfile}) \wedge (D \in \mathsf{CompoundProfile}) \wedge (\forall i \in K_{D.\mathsf{NamedProfiles}} \mid ((C \propto_{\text{simpleProfile}} D.namedProfiles_i.simpleProfile) \vee NotReachable(D.namedProfiles_i, D))) \wedge (\forall j \in K_{D.\mathsf{Transitions}} \mid transition_j.ops = \emptyset)$$

We have excluded the possibility for any transitional behaviour in the compound profile. This may in some cases be too restrictive since one can have operations that would have no effect on the perceived behaviour of the component, e.g., instantaneous notifications that take no perceivable time to complete and have no side-effects. However, operational semantics is not available, it is therefore not possible to identify such operations and we therefore have to exclude all operations. Such operational semantics may be available in the future, and the conformance rules can be changed accordingly. We have also excluded the profiles in the compound profile that are not reachable, i.e., those that have profiles prior in the precedence with weaker restrictions on the environment. The simple profile *C* may not conform to such profiles, but since they can never be used by any component, they can therefore be excluded from conformance considerations. We have used the predicate *NotReachable*(*namedProfile* ∈ NamedProfile, *compound* ∈ CompoundProfile) to denote this, it is defined as:

$$
\begin{aligned}
&\mathit{NotReachable}(\mathit{namedProfile} \in \textsf{NamedProfile}, \mathit{compound} \in \textsf{CompoundProfile}) \Leftrightarrow \\
&\quad \exists i \in K_{\mathit{compound}.\textsf{Precedences}}, \exists \mathit{profile} \in \mathit{compound}.\textsf{NamedProfiles} \mid (\\
&\qquad (\mathit{profile.profileName} = \mathit{compound.precedences}_i) \wedge (\mathit{profile} = \mathit{namedProfile}) \\
&\qquad \wedge (\exists j{<}i \in K_{\mathit{compound}.\textsf{Precedences}}, \exists \mathit{otherProfile} \in \mathit{compound}.\textsf{NamedProfiles} \mid (\\
&\qquad\quad (\mathit{otherProfile.profileName} = \mathit{compound.precedences}_j) \wedge \\
&\qquad\quad (\mathit{otherProfile.simpleProfile.uses} \propto_{\text{uses}} \mathit{namedProfile.simpleProfile.uses})\ )\ ) \\
&\quad)
\end{aligned}
$$

where *compound*.NamedProfiles is the set of all named profiles in *compound*:

$$\mathit{compound}.\textsf{NamedProfiles} = \bigcup_{i \in K_{\mathit{compound}.\textsf{NamedProfiles}}} \{\ \mathit{compound.namedProfiles.namedProfile}_i\}$$

Note that this predicate uses the fact that profiles are totally ordered in the precedence list. If other transition rules are used (such as a state machine) to determine what profile to activate in the case of a violation of the current expectation, the predicate above needs to be redefined accordingly.

4.7.2.1.3 Compound Profile Conforms to Simple Profile

We have now specified conformance when *C* is a simple profile and continue with the case that *C* is a compound profile. If a compound profile is to conform to a simple one, the compound profile has to include a reachable profile that conforms to the simple profile. Furthermore, all reachable profiles that have expectations that conform to the expectations of the simple profile, must provide offers that conform to the offer of the simple profile. Formally, that the compound profile *C* conforms to the simple profile *D* can be defined as:

$$
\begin{aligned}
C \propto_{\mathit{compoundProfile}} D \Leftarrow\ & (C \in \textsf{CompoundProfile}) \wedge (D \in \textsf{SimpleProfile}) \wedge \\
& (\exists i \in K_{C.\textsf{NamedProfiles}} \mid (C.\mathit{namedProfiles}_i.\mathit{simpleProfile} \propto_{\text{simpleProfile}} D \wedge \\
& \qquad\qquad \neg\, \mathit{NotReachable}(C.\mathit{namedProfiles}_i, C)\ )\ ) \wedge \\
& (\forall j \in K_{C.\textsf{NamedProfiles}} \mid \\
& \quad (C.\mathit{namedProfiles}_j.\mathit{simpleProfile.uses} \propto_{\mathit{uses}} D.\mathit{uses} \wedge \\
& \quad\ \neg\, \mathit{NotReachable}(C.\mathit{namedProfiles}_i, C)\ ) \Rightarrow \\
& \qquad C.\mathit{namedProfiles}_j.\mathit{simpleProfile.provides} \propto_{\text{provides}} D.\mathit{provides})
\end{aligned}
$$

4.7.2.1.4 Two Compound Profiles

The last case of profile conformance is when both QoS profiles are compound profiles. If *C* and *D* are compound profiles, *C* conforms to *D* iff each simple profile in *C* conforms to a simple profile in *D* and the transitions and precedences in *C* conforms to the transitions and precedences in *D*, respectively:

$$
\begin{aligned}
C \propto_{\mathit{compoundProfile}} D \Leftarrow\ & (C \in \textsf{CompoundProfile}) \wedge (D \in \textsf{CompoundProfile}) \wedge \\
& (\forall pC \in C.\textsf{NamedProfiles}, \exists\, pC \in D.\textsf{NamedProfiles} \mid (\\
& \quad (pC.\mathit{simpleProfile} \propto_{\text{simpleProfile}} pD.\mathit{simpleProfile}) \wedge \\
& \quad (\mathrm{C}.\mathit{transitions} \propto_{\text{transitions}} D.\mathit{transitions}) \wedge \\
& \quad (C.\mathit{precedences} \propto_{\text{precedences}} D.\mathit{precedences}\ )\ )\ )
\end{aligned}
$$

A list of transitions *E* conforms to another list of transitions *F* iff each single transition in *F* has a single transition in *E* that conforms to it, i.e., that the list *E* contains at least a list of transitions that conform to *F*:

$$E \propto_{\text{transitions}} F \Leftrightarrow \forall \mathit{ftrans} \in F.\textsf{Transition}, \exists \mathit{etrans} \in E.\textsf{Transition} \mid \mathit{etrans} \propto_{\text{transition}} \mathit{ftrans}$$

where *F*.Transition denotes the set of all single transitions in *F*:

$$F.\textsf{Transition} = \bigcup_{i \in K_{F.\mathit{transitions}}} \{F.\mathit{transitions}_i\}$$

A single transition consists of the names of a from- and a to-profile together with operations to perform when a transition occurs. For two transitions to conform, the from- and to-profile names in both transitions must name profiles that conform, respectively, and the operations must conform. Hence, if a transition G conforms to a transition H, the from- and to-profile names in a transition *G* conforms to the from- and to-profile names in a transition *H*, respectively, and the operations in *G* conforms to the operations in *H*:

$$\begin{aligned} G \propto_{\text{transition}} H \Leftrightarrow\ & (G.from \propto_{\text{from}} H.from) \wedge (G.to \propto_{\text{to}} H.to) \wedge \\ & (G.operations \propto_{\text{operations}} H.operations) \end{aligned}$$

A from-profile name conforms to another from-profile name iff the profiles they name conform. Formally, a from-profile name $I = G.from$ conforms to a from-profile name $J = H.from$ if I is $\{any\}$ or there exists a simple profile named I in the compound profile in which I is part that conforms to a simple profile named J in the compound profile in which J is a part:

$$\begin{aligned} I \propto_{\text{from}} J \Leftrightarrow\ & (I = any) \vee \\ & (\exists iprofile \in \mathsf{NamedProfile}_I \mid (\\ & \quad (iprofile.profileName = I) \wedge \\ & \quad (\exists jprofile \in \mathsf{NamedProfile}_J \mid ((jprofile.profileName = J) \wedge \\ & \qquad (iprofile.simpleProfile \propto_{\text{simpleProfile}} jprofile.simpleProfile))))) \end{aligned}$$

where $\mathsf{NamedProfile}_I$ denotes the set of named profiles in the compound profile in which I is a part.

Correspondingly, a to-profile name $K = G.to$ conforms to a to-profile name $L = H.to$ if either both are $\{any\}$ or K is not $\{any\}$ and there exists a simple profile named K in the compound profile in which K is part that conforms to a simple profile named L in the compound profile in which L is a part:

$$\begin{aligned} K \propto_{\text{to}} L \Leftrightarrow\ & (((K = any) \wedge (L = any)) \vee \\ & \quad ((K \neq any) \wedge \\ & \qquad (\exists kprofile \in \mathsf{NamedProfile}_K \mid (\\ & \qquad\quad (kprofile.profileName = K) \wedge \\ & \qquad\quad (\exists lprofile \in \mathsf{NamedProfile}_L \mid ((lprofile.profileName = L) \wedge \\ & \qquad\qquad (kprofile.simpleProfile \propto_{\text{simpleProfile}} lprofile.simpleProfile)))))) \\ & \quad) \end{aligned}$$

Two lists of operations conform if their effects conform, i.e., they must produce the same results on those parts of the objects that are common. Formally, a list of operations *M* conforms to another list of operations *N* iff the effect of *M* on the history of the system equals the effect of *N* on the system history, except for those characteristics of the component type to which *N* applies that are not part of the component type to which *M* applies:

$$\begin{aligned} M \propto_{\text{operations}} N \Leftrightarrow\ & (History(M) - History(@M)) = \\ & (History(N)\backslash(componentType_N - componentType_M) - \\ & \quad History(@N)\backslash(componentType_N - componentType_M)) \end{aligned}$$

where $History(M)$ denotes all histories (the global history) after invocation of the operation in M, $History(@M)$ denotes the histories prior to invocation of the operations in M, and $History(N)\backslash(componentType_N - componentType_M)$ denotes the system history except those parts that are part of the component type to which *N* applies but not in the component type to which *M* applies, after invoking *N*.

A profile precedence conforms to another iff the ordering in the other is preserved, i.e., a precedence *O* conforms to another precedence *P* if all profiles that are named in *P* have conforming and reachable profiles named in *O* in the same order:

$$\begin{aligned}
O \propto_{\text{precedence}} P \Leftrightarrow\ & \forall i,j \in K_P \mid (\\
& \quad (i < j) \Rightarrow (\exists m,n \in K_O \mid (\\
& \qquad (m < n) \wedge \\
& \qquad (O_m.profile.simpleProfile \propto_{\text{simpleProfile}} P_i.profile.simpleProfile) \wedge \\
& \qquad (O_n.profile.simpleProfile \propto_{\text{simpleProfile}} P_j.profile.simpleProfile) \wedge \\
& \qquad \neg NotReachable(O_n.profile) \wedge \neg NotReachable(O_m.profile))))
\end{aligned}$$

where K_P denotes the index set of P and $O_m.profile$ is a shorthand for the simple profile that is named O_m:

$$O_m.profile = p \in \mathsf{NamedProfile}_O \mid p.profileName = O_m$$

We have now defined conformance for QoS profiles under the assumption that there exists a conformance relation for QoS statements. Therefore, we address this next.

4.7.2.2 QoS Statements

A QoS statement A conforms to a QoS statement B (denoted $A \propto_{\text{statement}} B$) if the formal parameters and bindings in A conforms to the formal parameters and bindings in B, respectively, and for each constrained QoS characteristic in B, there exists a constrained QoS characteristic in A that conforms to the one in B:

$$\begin{aligned}
A \propto_{\text{statement}} B \Leftrightarrow\ & (A.qosStatFormalParams \propto_{\text{qosStatParams}} B.qosStatFormalParams) \wedge \\
& (A.qosStatBindings \propto_{\text{qosStatBindings}} B.qosStatBindings) \wedge \\
& (\forall b \in B.\mathsf{ConstrQoSChar}, \exists a \in A.\mathsf{ConstrQoSChar} \mid a \propto_{\text{constrQoSChar}} b\ )\ )
\end{aligned}$$

where $B.\mathsf{ConstrQoSChar}$ denotes the set of constrained QoS characteristics in B:

$$B.\mathsf{ConstrQoSChar} = \bigcup_{i \in K_{B.constrQoSChars}} \{B.constrQoSChars_i\}$$

A list of formal parameters C conforms to another list D if the type of each parameter in D conforms to a parameter in C:

$$C \propto_{\text{qosStatParams}} D \Leftrightarrow \forall i \in K_D, \exists j \in K_C \mid C_j \propto_{\text{qosStatParam}} D_i$$

Since a formal parameter is a named type, we use type conformance as discussed previously to define that two formal E and F parameters conform:

$$E \propto_{\text{qosStatParam}} F \Leftrightarrow E.type \propto_{\text{type}} F.type$$

The definition of conformance between the bindings in QoS statements corresponds to the definition of conformance between bindings in QoS profiles in subsubsection 4.7.2.1. We do therefore not repeat that discussion, but only provide the definition. If we denote two bindings G and H, that G conforms to H is defined as:

$$\begin{aligned}
G \propto_{\text{qosStatBindings}} H \Leftrightarrow\ & \forall sb_G \in G.\mathsf{SingleStatBinding}, \exists sb_H \in H.\mathsf{SingleStatBinding} \mid (\\
& \quad (\ sb_G.apchar \propto_{\text{qosCharParam}} sb_H.apchar\ ) \wedge \\
& \quad (\ sb_G.fpstat \propto_{\text{qosStatParam}} sb_H.fpstat\ )\ )
\end{aligned}$$

A constrained QoS characteristic I conforms to a constrained QoS characteristic J if the QoS characteristic of I conforms to the one of J, if the constraining predicate of I is at least as restricting as that of J, and that the discriminator of I conforms to the discriminator of J:

$$\begin{aligned}
I \propto_{\text{constrQoSChar}} J \Leftrightarrow\ & (I.qosChar \propto_{\text{qosChar}} J.qosChar) \wedge (QoSStatPred_I \Rightarrow QoSStatPred_J) \wedge \\
& (I.discriminator \propto_{\text{discriminator}} J.discriminator)
\end{aligned}$$

A discriminator K conforms to a discriminator L if K is $\{guaranteed\}$ or if their qualifiers conform.

$$K \propto_{\text{discriminator}} L \Leftrightarrow (K = guaranteed) \vee (K.qualifier \propto_{\text{qualifier}} L.qualifier)$$

That a qualifier *M* conforms to a qualifier *N* means that they must be of the same type and the limit value of *N* must be less than or equal to that of *M*, i.e., *M* can only constrain *N* further:

$$M \propto_{\text{qualifier}} N \Leftrightarrow (M.type = N.type) \wedge ((N.value <_{\text{result}} M.value) \vee (N.value = M.value))$$

where *M.type* denotes the qualification type (threshold or compulsory) and *M.value* denotes the limit value of the qualification. Note that the limit values are compared using the ordering operator $<_{\text{result}}$, this ensures that the semantic direction is included in the comparison.

We have now defined conformance between QoS statements under the assumption that their constituent QoS characteristics have a conformance relation defined. We address this next.

4.7.2.3 QoS Characteristics

A QoS characteristic *A* conforms to another QoS characteristic *B* if the value space of *A* is within the value space of *B* and if all orderings and specifications of the values of *A* correspond to orderings and specifications of the values of *B*. Formally, a QoS characteristic *A* conforms to a QoS characteristic *B* (denoted $A \propto_{\text{qosChar}} B$) if the following applies:

$$\begin{aligned} A \propto_{\text{qosChar}} B \Leftrightarrow\ & (A.qoSCharDomain \subseteq B.qoSCharDomain) \wedge (A.pred \Rightarrow B.pred) \\ & \wedge (B.stat \subseteq A.stat) \wedge (A.<_{\text{result}} \subseteq B.<_{\text{result}}) \wedge \\ & (A.formalParams \propto_{\text{qosCharParams}} B.formalParams) \wedge \\ & (A.valueSpec \subseteq B.valueSpec) \wedge (B.comp \subseteq A.comp) \end{aligned}$$

In other words, a QoS characteristic *A* conforms to a QoS characteristic *B* if:

- the value domain of *A* is a subdomain of the value domain of *B*,
- the statistical aspects of *B* are also specified in *A*,
- the ordering of values in *B* is preserved in the ordering of values in *A*, i.e., those values in *B* that also are in *A* have the same ordering in both *A* and *B* under $<_{\text{result}}$,
- the parameters for *A* conform to the ones for *B*,
- the mapping from parameters to the value domain in *B* is preserved for those domain values that are part of *A*, and
- the three different composition mappings are preserved for those domain values in *B* that are part of *A*.

Note that *valueSpec* and any of the three composition functions in *comp* can be *Undefined*. For this, the following rule applies:

$$Undefined \in \{X\}$$

where *X* denotes any mapping.

Note also that the subset relation between orderings uses the fact that such orderings are transitive, so that any two ordered values in the domain of *A* will have the same ordering in *B*:

$$\forall x,y \in A.qoSCharDomain \mid x\ A.<_{\text{result}}\ y \Rightarrow x\ B.<_{\text{result}}\ y$$

The same consideration applies for value specifications that effectively are pairs of formal parameter values and a value in the domain of the QoS characteristic:

$$\begin{aligned} & \forall x \in A.formalParams, \exists y \in B.formalParams, \forall z \in A.qoSCharDomain \mid \\ & \qquad x\ A.valueSpec\ z \Rightarrow (y\ B.valueSpec\ z \wedge x \propto_{\text{qosCharParams}} y) \end{aligned}$$

It also applies for compositions:

$$\begin{aligned} & \forall x,y,z \in A.qoSCharDomain \mid \\ & \qquad (x\ A.comp.compFunc_{\text{parallel-or}}\ y = z \Rightarrow x\ B.comp.compFunc_{\text{parallel-or}}\ y = z\ ) \wedge \\ & \qquad (x\ A.comp.compFunc_{\text{parallel-and}}\ y = z \Rightarrow x\ B.comp.compFunc_{\text{parallel-and}}\ y = z) \wedge \\ & \qquad (x\ A.comp.compFunc_{\text{sequential}}\ y = z \Rightarrow x\ B.comp.compFunc_{\text{sequential}}\ y = z) \end{aligned}$$

Conformance between formal parameters is equal to conformance between formal parameters for QoS statements:

$$\propto_{\text{qosCharParams}} = \propto_{\text{qosStatParams}}$$
$$\propto_{\text{qosCharParam}} = \propto_{\text{qosStatParam}}$$

It is important to note that a specialised QoS characteristic is not necessarily conformant to the one it specialises. A specialised QoS characteristic may define, for instance, a new value specification that conflicts with the one it specialises, possibly by using a formal parameter that is only part of this specialised QoS characteristic.

4.7.3 QoS Contracts and Validity of QoS Profiles

A QoS contract is an agreement between two collaborating components that primarily specifies the level of QoS one of the parties agrees to provide to the other. However, since it is sometimes also required that the client provides some services in return for a collaboration to take place, e.g., when callback handlers are required, a QoS contract may specify the levels of QoS both parties agree to provide to each other. A collaboration is initiated based upon one party providing some services that another requires. However, such a collaboration may only happen if the constraints specified in the QoS profiles also match. A QoS contract is established if the level of QoS offered by one is sufficient to meet some of the requirements of the other. In other words, a QoS profile *A* may form a QoS contract with a QoS profile *B* if there exists a conjunct in the provides-clause of *B* that conforms to a conjunct in the uses-clause of *A*. If we use $\leftsquigarrow\!\!\rightsquigarrow$ to denote such a partial match, we define $A \leftsquigarrow\!\!\rightsquigarrow B$ as:

$$A \leftsquigarrow\!\!\rightsquigarrow B \Leftrightarrow \exists \mathit{providesConjunct} \in B.\mathsf{ProvidesConjuncts}, \exists \mathit{usesConjunct} \in A.\mathsf{UsesConjuncts} \mid \\ (\mathit{providesConjunct} \propto_{\text{conjunct}} \mathit{usesConjunct}) \wedge \\ \mathit{usesConjunct.invariant} \wedge \mathit{providesConjunct.invariant}$$

where $B.\mathsf{ProvidesConjuncts}$ is the set conjuncts in the provides-clause of all simple profiles of *B*:

$$B.\mathsf{ProvidesConjuncts} = \left(\bigcup_{i \in K_{B.profileBody.simpleProfile.provides}} \{B.profileBody.simpleProfile.provides_i\} \right) \cup \\ \{\forall j \in K_{B.profileBody.compoundProfile.namedProfiles} \mid \left(\bigcup_{k \in K_{B.profileBody.compoundProfile.namedProfiles_j.namedProfile.SimpleProfile.provides}} \\ B.profileBody.compoundProfile.namedProfiles_j.namedProfile.simpleProfile.provides_k \right) \}$$

Correspondingly, $A.\mathsf{UsesConjuncts}$ is the set of all conjuncts in the uses-clause of all simple profiles of *A*.

usesConjunct.invariant denotes the invariant of the simple profile of which *usesConjunct* is a part.

The partial match defined here depends on valid QoS profiles, i.e., QoS profiles where an offered QoS can be provided. We assume that QoS specifications are truthful, i.e., that, to the best of the specifier's ability, a QoS offer can be provided if its QoS expectations are met. Hence, a QoS profile is valid if there exist valid QoS profiles that can fulfil its QoS expectations. Matching of QoS profiles is a job for the QoS framework. For this, it may keep track of the validity of provided QoS offers. Invalid offers should not be able to be a part of negotiations to establish QoS contracts. Hence, the partial match defined above only specifies a potential collaboration between two components. A component with a profile *A* can provide the QoS offer specified in *A* if there exist components in its environment with QoS profiles that collectively meet all of its expectations. We define a predicate $Valid(x \in \mathsf{QoSProfile})$ that specifies whether a profile contains valid QoS offers:

$$
\begin{aligned}
\mathit{Valid}(x \in \text{QoSProfile}) \Leftrightarrow\ & \forall \mathit{usesConjunct} \in x.\text{UsesConjuncts}\ ,\ \exists \mathit{profile} \in \text{QoSProfile} \mid \\
& \exists \mathit{providesConjunct} \in \mathit{profile}.\text{ProvidesConjuncts}, \mid (\\
& \quad (\mathit{providesConjunct} \propto_{\text{conjunct}} \mathit{usesConjunct}) \wedge \\
& \quad \mathit{usesConjunct.invariant} \wedge \mathit{providesConjunct.invariant} \wedge \\
& \quad \mathit{Valid}(\mathit{profile})\ )
\end{aligned}
$$

This recursive definition of the validity of QoS profiles shows that to decide whether a QoS profile is valid requires validity evaluation of the transitive closure of those QoS profiles that may provide the expected properties. Such a transitive closure will eventually end up with offers that either have no expectations or that have their expectations met by measurements at run-time. Whether such expectations are met cannot be determined until run-time when measurements can be made, so the QoS framework can only use some nominal values for these expectations to determine validity. However, these considerations are not within the scope of CQML, it only provides the information that can be used by the QoS framework for this purpose.

4.7.4 Relaxed Conformance

Substitutability requires strict conformance, i.e., that all aspects of a component specification conform to the one it is substitutable with. In some cases, a QoS framework may not be able to find a single conformant component but may still want to perform substitution. This is for instance the case if a component no longer is able to provide its QoS offer and the policy is to maintain the service level whatever the cost. In such a case, the QoS framework is primarily concerned with conformance of the QoS offers. Conformance of the QoS expectations (which essentially represent cost) is only of secondary interest to enable that the less costly of the components that has conforming QoS offers, can be chosen. Correspondingly, in some cases the QoS framework may want to focus on conformance of the QoS expectations and be less concerned with conformance of the QoS offers. This is for instance the case if the resource consumption should be kept under control to avoid resource shortages and an inflationary situation. The policy of the QoS framework may then be to try to make the best out of the requested resources (as specified in the QoS expectation). In this situation, the QoS framework may want to replace a component with another one that uses the same (or less) resources, but that potentially not provides a conformant QoS offer.

Formalisations of both offer-conformance and expectation-conformance are straightforward specialisations of the formalisation of strict conformance where only $\propto_{\text{provides}}$ and $\propto_{\text{uses}}$ are considered, respectively. The details are omitted here.

Such relaxed conformance may be used during both adaptation and service initialisation. For instance, it may not be possible to configure a system so that all required QoS contracts can be established with valid QoS profiles unless one uses relaxed conformance for some of the QoS contracts.

4.7.5 Composition of QoS Profiles

As discussed in subsubsection 4.6.5.4, the composition theorem by Abadi and Lamport applies to QoS profiles if the QoS statements are specifying safety properties. The theorem, as adapted to QoS relations in [Leboucher and Najm, 1997], specifies that two QoS relations `Exp(O`$_1$`)` $\mapsto$ `Obl(O`$_1$`)` and `Exp(O`$_2$`)` $\mapsto$ `Obl(O`$_2$`)`, where `Exp(O`$_i$`)` is a QoS expectation (assumption) and `Obl(O`$_i$`)` is a QoS offer (guarantee), can be composed using the inference rule

$$\frac{Exp \cap Obl(O_1) \cap Obl(O_2) \subseteq Exp(O_1) \cap Exp(O_2)}{\big(Exp(O_1) \mapsto Obl(O_1)\big) \cap \big(Exp(O_2) \mapsto Obl(O_2)\big) \subseteq Exp \mapsto Obl(O_1) \cap Obl(O_2)}$$

Exp denotes supplementary constraints on the environment of the two objects. The inference rule is sound if the specifications are safety properties (such as if `Exp(O_i)` does not constrain O_i, and `Obl(O_i)` at most constrains O_i).

In other words, the composition theorem specifies that if the conjunction of the QoS offers for O_1 and O_2 implies the conjunction of the QoS expectations, then O_1 and O_2 are composable and the resulting QoS offer is the conjunction of the individual QoS offers.

Note that the assumption in the inference rule may constrain when the individual QoS profiles are valid if they are to be parts of the composition, i.e., the *Valid*-predicate defined in subsection 4.7.3 needs to take these constraints into consideration if the components are parts of compositions. Note also that the composition theorem does not apply to all kinds of composition, only to the important category of conjoining compositions.

4.8 Example

In this section, we provide the running example used to illustrate the concepts in CQML in its entirety. A more in-depth discussion of how CQML can be used in software development is provided in Chapter 5.

4.8.1 Video Camera

In this subsection, we refine the example introduced in subsubsection 4.6.5.1, which in turn is based on an example in [Blair and Stefani, 1997]. The design consists of a video camera, a video presentation window, and a QoS manager that monitors the achieved QoS. Between these objects we specify object communication bindings, a video stream binding and a QoS binding. We treat such bindings as first class objects. We also specify a camera operator that operates (start, stop, etc.) the video camera. We do not specify a binding object for this interaction, we assume local communication using an implicit binding. Using UML, the design is shown in Figure 20. The associations show which objects that interact and the dependencies show which interfaces the objects provide or require.

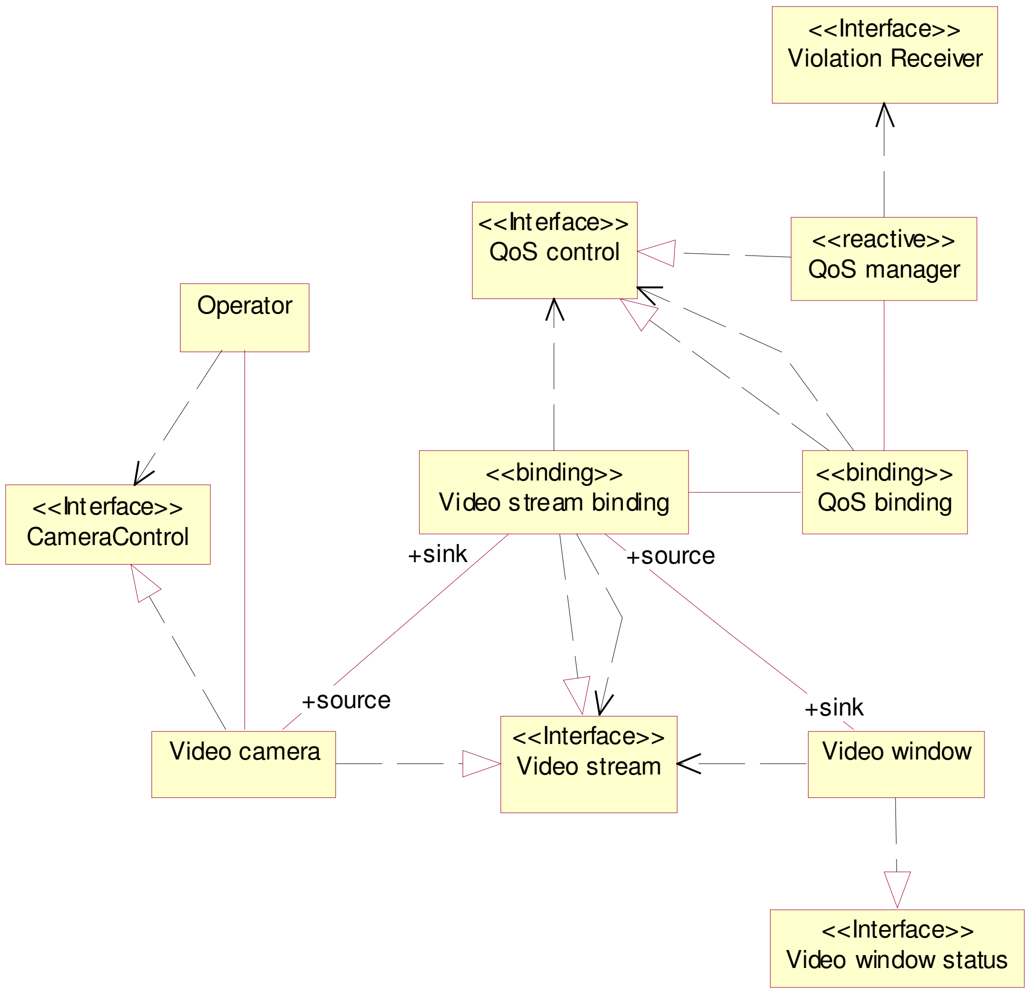


Figure 20: Simple video application example

The video camera implements the video stream interface while the video window requires it. The video stream binding acts as an intermediary, both implementing and providing the video stream interface. The same constellation is used for the QoS manager. Note that the QoS manager is stereotyped as a reactive object meaning that it has deterministic time behaviour. The bindings are stereotyped as bindings.

In this design, interesting QoS characteristics are output rates of the video camera and the video window, and delay of the binding between the camera and the video window. In addition, we want to specify that the QoS manager is notified of the relevant events in the system so that it can take appropriate actions when needed.

The basic operation consists of a video camera that delivers a video flow to a binding object, which then delivers it to a video presentation window. The binding object sends signals each time a video frame is sent from the video camera and each time a video frame is delivered to the presentation window. The QoS manager uses these signals to monitor the performance of the application. We specify a camera operator that can perform camera-controlling operations.

The stream interface specification for the output of a video camera was introduced in subsubsection 4.6.5.2:

```
interface VideoStream {
  Video videoFlow;
};
```

The video camera has also some controlling operations. It is possible to start and stop the camera, and perform a number of manipulative operations like pan, tilt and zoom. We specify this as an ordinary operational interface:

```
interface CameraControl {
  void start();
  void stop();
  void pan(in short panDegrees);
  void tilt(in short tiltDegrees);
  void zoom(in short zoomFactor);
};
```

We have specified two interfaces, both of which relate to video camera. When specifying the video camera, we want to specify that the camera should emit at least 25 frames per second and that it should start within 10 milliseconds from when a start-operation is received. Using the `eventsInRange`-operation defined on `EventSequence` in subsubsection 4.6.5.2, we specify the QoS characteristic `frameOutput` as:

```
quality_characteristic frameOutput (flow : Flow) {
  domain: increasing numeric frames/second;
  values: flow.SE->eventsInRange(1000);
}
```

A QoS statement `goodVideo` can then be defined as:

```
quality goodVideo (flow : Flow) {
  frameOutput(flow) >= 25;
}
```

Also, to specify that the camera should start within 10 milliseconds from when a start-operation is received, we need the QoS characteristic `startUpTime`. From its definition in subsubsection 4.6.3.4, we add the initiating event as a parameter since we model the initiating event as the `start`-operation in the `CameraControl` interface:

```
quality_characteristic startUpTime (
             flow : Flow; initiatingEvent : Event) {
  domain:  decreasing numeric milliseconds;
  values:  if flow.SE->isEmpty then invalid
           else flow.SE->first.time() - flow.initiate.time()
           endif;
  invariant: flow.initiate = initiatingEvent;
}
```

We use the `initiate`-attribute of the `videoFlow` object to identify the time of initiation of the flow. As the invariant defines, this attribute must be set to be the event when the camera object receives the `initiatingEvent` (i.e., the time “Play” is pressed on the camera and the `start`-operation is invoked) and a new `Flow` object is created.

A QoS statement `fast` can then be defined as:

```
quality fast (flow : Flow; initiator : Operation) {
  startUpTime(flow, initiator.SR->last) <= 10;
}
```

Note that we extract the initiating event in this QoS statement (`initiator.SR->last`). The history of the object projected onto its initiator operation will grow as new initiations occur. So whenever the QoS characteristic `startUpTime` is evaluated, we need the event that caused this particular instance of the flow to start emitting frames. If we had postponed the extraction of the proper event until each evaluation of `startUpTime`, the last event in the sequence of initiations could have changed by any following `Play`-messages being received, violating the invariant.

As mentioned in subsection 4.6.5, we specify components that implement interfaces using CIDL:

```
component VideoCamera {
  provides CameraControl control;
  provides VideoStream out;
};
```

We can now specify a QoS profile for the `VideoCamera` component:

```
profile goodCamera for VideoCamera {
  provides goodVideo(out.videoFlow) and
           fast(out.videoFlow, control.start);
}
```

For the model to be complete, we specify an operator that operates the video camera. An operator is an active object that initiates activities through the control interface on the video camera. We use the same interface definition as for the camera to specify the operator, except that it is an incoming interface for the operator:

```
component operator {
  uses CameraControl control;
};
```

The video camera specified above delivers a service and provides some characteristics independent of the underlying infrastructure. In order for a video camera to be useful, for instance in a surveillance application, the video flow must be displayed. For an object to have displaying capabilities, two interfaces must be supported. First, it must be able to receive a video flow and display it. Second, it should emit a signal for each frame it presents so that a QoS manager can monitor if the end-user QoS is acceptable. We want a frame to be presented within 10 milliseconds from when the frame is received.

The signalling interface of the video window can be defined as:

```
interface VideoWindowStatus {
  Event videoPresented;
};
```

We use the `VideoStream` interface defined earlier to receive the video. The video window component can then be defined as:

```
component videoWindow {
  provides VideoWindowStatus status;
  uses VideoStream incoming;
};
```

We want to be able to specify that a frame is presented within 10 milliseconds from when it was received, we therefore need to generalise the `delay` characteristic defined in subsubsection 4.6.5.2 to handle `EventSequence` instead of `Flow`:

```
quality_characteristic delay(
     firstSeq : EventSequence; secondSeq : EventSequence) {
  domain: decreasing numeric milliseconds;
  values: firstSeq->maxTimeDifference(secondSeq);
}
```

Note that we here define delay as the maximum time difference between any two “equal” events in any two event sequences, not only two events from different event sequences that we know are from the same location. However, the identity operator used in the definition of `maxTimeDifference` ensures that if the event sequences are not in the same context (e.g., use the same clock), the result is undefined.

This new definition of delay can be used to specify the maximum delay of notification of frame reception:

```
quality quickPresentation (incoming : Flow;
                           status : EventSequence) {
  delay(incoming.SR, status) <= 10;
}
```

Also, we need to modify the `goodVideo` QoS statement as we need to use the signals that the video window emits when presenting a frame instead of the flow itself. An appropriate QoS characteristic can be defined as:

```
quality_characteristic framesShown (seq : EventSequence) {
  domain: increasing numeric frames/second;
  values: seq->eventsInRange(1000);
}
```

The QoS statement `goodVideoShown` can then be defined as:

```
quality goodVideoShown (seq : EventSequence) {
  framesShown(seq) >= 25;
}
```

We are now in the position to define a QoS profile for the video window:

```
profile nice for  videoWindow {
  uses      goodVideo(incoming.videoFlow);
  provides  goodVideoShown(status.videoPresented.SE) and
            quickPresentation(incoming.videoFlow,
                              status.videoPresented.SE);
}
```

We have now specified that the video window shows at least 25 frames per second, and that it shows each frame within 10 milliseconds from when it is received. Together with the specification of the video camera above, we have specified a producer and a consumer with some requirements. The requirements are that the video flow should be delivered to the consumer with the same quality constraints as is put on the producer. We must therefore specify the binding object between the producer and the consumer. A main QoS characteristic of an object communication binding is its delay. We want this to be under 60 milliseconds. However, we do not want the delay to vary too much since that reduces the video quality, so we want it to have a variation of maximum 20 milliseconds. Instead of specifying a separate characteristic for this jitter, we simply constrain the `delay` characteristic with both upper and lower boundaries. Also, since we want the delay to be between 40 and 60 milliseconds, we want its mean to be 50 milliseconds. This is specified as:

```
quality fastAndSteadyTransfer (  seq1 : EventSequence;
                                 seq2 : EventSequence) {
  delay(seq1, seq2).maximum <= 60;
  delay(seq1, seq2).minimum >= 40;
  delay(seq1, seq2).mean = 50;
}
```

We want to make sure that the binding tells the QoS manager of how it is performing by emitting appropriate signals, we therefore define the `QoSControl` interface.

```
interface QoSControl {
  Event videoSent;
  Event videoDelivered;
};
```

To make sure that the signals are timely emitted, we specify the time difference between emitting signals and the event that causes this to be less than 1 millisecond:

```
quality instantaneous ( cause : EventSequence;
                        effect : EventSequence) {
  delay(cause, effect) < 1;
}
```

We treat bindings that require complex binding actions as first class objects, and can therefore specify the video binding as:

```
component videoBinding {
  provides VideoStream outgoing;
  provides QoSControl control;
  uses VideoStream incoming;
};
```

We are now in the position to specify a QoS profile for the binding:

```
profile fastAndSteady for videoBinding {
  uses goodVideo(incoming.videoFlow);
  provides goodVideo(outgoing.videoFlow) and
          fastAndSteadyTransfer(incoming.videoFlow.SR,
                                outgoing.videoFlow.SE) and
          instantaneous(incoming.videoFlow.SR,
                        control.videoSent.SE) and
          instantaneous(outgoing.videoFlow.SE,
                        control.videoDelivered.SE);
}
```

We specify in this profile that the video binding provides a number of QoS offers given that it gets a good video (i.e., with a frame rate of at least 25 frames per second) as input. It offers a good video (defined as for the input flow), it offers a fast and steady transfer (i.e., with a delay between 40 and 60 milliseconds with 50 milliseconds on average), and it reports on the events of receiving and sending video frames instantaneously (i.e., within 1 millisecond). It may seem like we now have only moved the problem of frame deliverance from between consumer and producer to between binding and producer as the binding has the same input requirement on frame delivery as the consumer. We have, however, introduced the `QoSControl` interface that can be used to monitor the QoS of the binding.

In order to manage timing constraints, we must end up in synchronous (reactive) objects. Due to the deterministic nature of reactive objects (they respond instantaneously), we can use reactive objects to anchor the timing dependencies down to deterministic behaviour, and thereby use these as managers without themselves having to be managed. Here we specify a reactive object with two signal interfaces, one for signals into the object and the other for emitting exceptions. We define a new type `ViolationEvent` that has an attribute specifying what type of violation that occurs, and we use this in the QoS Monitor interface that the QoS manager provides.

```
interface ViolationEvent : Event {
  integer violationType;
};

interface QoSViolation {
  ViolationEvent qosViolation;
};

component qosManager {
  provides QoSViolation violation;
  uses QoSControl control;
};
```

Note that the reactive object `qosManager` supports only signal interfaces. Since it is a reactive object, its execution (signalling `qosViolation` when this occurs) should be instantaneous and it has therefore no related QoS profile. Even though no execution happens instantaneously in real life, we assume that the time it takes is negligible.

We have now specified a flow producer and consumer, and the binding between them. We have also specified a reactive object to be used as QoS manager. The missing specification is the binding between the video binding and the QoS manager:

```
quality fastSignalDelivery ( seq1 : EventSequence;
                             seq2 : EventSequence) {
  delay(seq1, seq2) <= 5;
}

interface QoSMonitor {
  short averageDelay();
};

component qosBinding {
  provides QoSControl outControl;
  provides QoSMonitor monitor;
  uses QoSControl inControl;
};

profile fast for qosBinding {
  provides fastSignalDelivery(inControl.videoSent.SR,
                              outControl.videoSent.SE) and
           fastSignalDelivery(inControl.videoDelivered.SR,
                              outControl.videoDelivered.SE);
}
```

The specification states that the QoS binding should take no more than 5 ms to deliver the signals between the video binding and the QoS manager. The average delay can be polled using the `qosMonitor` interface using the `averageDelay` operation.

4.9 Run-time Representation

As part of his work for the degree of MSc, Tom Christian Paulsen has implemented a CQML-compiler that parses CQML and generates a run-time representation of the constructs of CQML [Paulsen, 2001]. Paulsen used an early version of CQML (v0.9) as a basis for his work, so the compiler does not handle all constructs described herein. The differences are:

- v0.9 does not include the notion of composition of QoS characteristics.
- QoS profiles are not included in v0.9. The association between QoS properties and component specifications is instead specified as an extension to TINA-ODL.
- Facilities for adaptation (worth functions and compound QoS profiles with transition behaviour) are not supported by v0.9.
- There are some minor syntactic variations (e.g., **`values`** is termed `semantics` in v0.9).

The CQML compiler builds a parse tree (using the JJTree parse tree builder for JavaCC). In this parse tree, appropriate code is included to export CQML specification into a QoS repository that stores run-time representations of CQML constructs. This QoS repository provides an interface for storing CQML constructs that the compiler uses when traversing the parse tree to populate the QoS repository. Figure 21 illustrates the process of populating the repository.

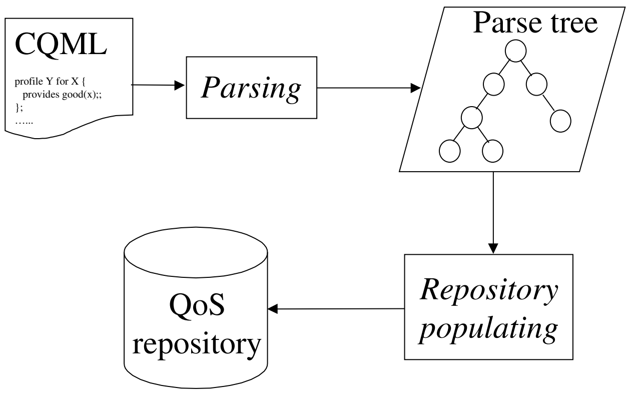


Figure 21: Generation of run-time representation

In addition to operations for storing, manipulating and removing CQML constructs from the QoS repository, the QoS repository also provides operations for checks of conformance and partial conformance.

The IDL for the QoS repository is provided in appendix I.

4.10 Summary

In this chapter, we defined the Component Quality Modelling Language (CQML). It is a lexical language for specifying QoS based on an ODP-compliant computational model. The concepts of QoS profile, QoS statement and QoS characteristic are used when specifying QoS in CQML. Different kinds of QoS statements (guaranteed, threshold best-effort, compulsory best-effort) can be specified. Also, derived and specialised QoS characteristics can be specified in CQML. CQML includes support for composition and adaptation.

Chapter 5 CQML in Software Development

In appendix I, four papers that address software development and QoS are presented. Chapter 4 presented the CQML. This chapter puts these together by positioning CQML in the software development approaches presented in appendix I combined with software development approaches developed by the distributed information systems group at SINTEF. We use a Lecture-on-Demand (LoD) case study for this.

An extension to the OMODIS project (termed the OMODIS LoD project) is in progress in which a medical LoD prototype is being built to enable students at the University of Oslo Medical School to study medical procedures over the internet. We have therefore used LoD as a case study for including QoS in object-oriented software development. In this chapter, we present the results of this work by examples of how QoS, expressed in CQML, can be positioned in selected parts of the LoD case study. We generalise this by presenting a UML profile for QoS that incorporates the principles illustrated in the case study.

However, we start this chapter by showing that CQML can be used in other parts of software development than addressed by an object-oriented methodology. We do this by showing that CQML can be used in requirements engineering, to specify service level agreements, and in process assessment.

5.1 CQML in Requirements Engineering

Requirements engineering is an important part of software systems engineering and has received a great deal of attention in the recent years. Requirements engineering is concerned with elicitation, representation and validation of requirements of software-intensive systems in order to define the purpose of a proposed system and outline its external behaviour. Most techniques for requirements engineering deal with functional requirements only. A notable exception is the work by Lawrence Chung et al., where a framework for dealing with non-functional requirements (the NFR-framework) is proposed and used [Mylopoulos et al., 1992, Chung et al., 1994, Gross and Yu, 2000, Nixon, 2000]. The NFR-framework is used for guiding the design process to design systems satisfying the non-functional requirements. The NFR-framework follows a goal-oriented methodology, akin to Cockburn's approach with use cases [Cockburn, 1997]. Cockburn's goal-oriented use cases has been extended to also include non-functional requirements in [Lee and Xue, 1999].

The main objective of the NFR-framework is to support the system development process from high-level non-functional requirements to a design that satisfies these. As part of this, the NFR-framework represents non-functional requirements as a goal-graph structure. Goals can be decomposed and methods can be specified to denote goal "satisficing" (i.e., sufficiently satisfy). The NFR-framework provides a set of correlation rules and methods that the developer may use to decide how to "satifice" the goals.

Based on the presentation of how the NFR-framework represents non-functional requirements in [Chung et al., 1994], CQML can be used to represent the non-functional requirements in an NFR-framework as follows.

- Non-functional requirements in the NFR-framework are associated with a set of concepts (termed sorts). Sorts correspond to QoS characteristics in CQML.
- Sorts may be organised into a hierarchy, e.g., decomposition of sorts into subsorts, for instance decomposition of security into confidentiality, integrity and availability. This corresponds to derived QoS characteristics in CQML. In CQML, the QoS characteristic security can be defined as a derived QoS characteristic dependant on the confidentiality, integrity and availability QoS characteristics.

- In the NFR-framework, goals are related to sorts and represent certain states of these sorts (even if absolute limits are not specified for goals, the notion of "satisficed" goals is used instead). Goals correspond to QoS statements in CQML with the distinction that QoS statements use specific limits following the vagueness argument from section 2.2.
- Goals in the NFR-framework are associated with components (most notably, the system) by declaring a parameter to which the goal pertains. In CQML, this is achieved by parameters to QoS statements.
- Correlation links between goals represent whether a goal contributes positively or negatively to another goal. Derived QoS characteristics in CQML used in QoS statements indicate that such links exist, but CQML does not provide the facility to specify a positive or negative contribution. In CQML, however, one can specify the exact contribution in the values-clause of the derived QoS characteristic if each of the constituent QoS characteristics has well-defined values-clauses.
- Goals are organised in a goal structure in the spirit of AND/OR-trees. In CQML, QoS characteristics can be related by AND and OR operators in QoS statements, and QoS statements can be related by AND and OR operators in QoS profiles.

To summarise, CQML provides the necessary capabilities to represent non-functional requirements in the NFR-framework with the exception of positive or negative contributions of correlation links between goals. The straightforward approach for this would be to add two keywords that denote whether the individual QoS characteristics contribute positively or negatively to the derived QoS characteristic.

To use CQML in requirements engineering is useful if the requirements are used as a starting point for design activities such as those described in section 5.4. Using CQML in requirements engineering ensures that the non-functional requirements are specified in an appropriate format for further refinement in the design activities, and traceability between requirements and design artefacts can be achieved.

5.2 CQML for Service Level Agreements (SLAs)

Service level agreements (SLAs) are often used between IT providers and end-user organisations to define characteristics such as response-time and availability of the IT services provided. With the current trend on outsourcing, SLAs are important to ensure that the buyer/seller relationships can be regulated under some agreed terms. In an SLA, the services and performance levels that will be provided are specified, as well as the metrics that will measure those capabilities. Furthermore, an SLA may also include specifications of the measures that can be taken if violations of the agreed terms occur.

An SLA will be specific to each IT provider and customer. However, most SLAs contain the same type of information, and in [Sturm et al., 2000a], templates for service level agreements are presented. The following is the short version of a template for an internal service level agreement as presented in [Sturm et al., 2000b] (the template is customised by filling in all text in underlined italic):

"The *insert service name* is used by *insert description of user community* to *insert description of the service capability*. The IT department guarantees that

1. The *service name* will be available *insert percentage* of the time from *insert normal hours of operation including hours and days of the week*. Any individual outage in excess of *insert time period or sum of outages* exceeding *insert time period per month* will constitute a violation.
2. *Insert percentage* of *service name* transactions will exhibit *insert value seconds* or less response time, defined as the interval from the time the user sends a transaction to the time a visual confirmation of transaction completion is received. Missing the metric for business transactions measured over any business week will constitute a violation.
3. The IT *department* will respond to service incidents that affect multiple users within *insert time period*, resolve the problem within *insert time period*, and update status every *insert time period*. Missing any of these metrics on an incident will constitute a violation.
4. The IT department will respond to service incidents that affect individual users within *insert time period*, resolve the problem within *insert time period*, and *update* status every *insert time period*. Missing any of these metrics on an incident will constitute a violation.
5. The IT *department* will respond to non-critical inquiries within *insert time period*, deliver an answer within *insert time period*, and update status within *insert time period*. Missing any of these metrics on an incident will constitute a violation."

Figure 22: Template of Internal Service Level Agreement

The following model illustrates the relationships between the elements of the template.

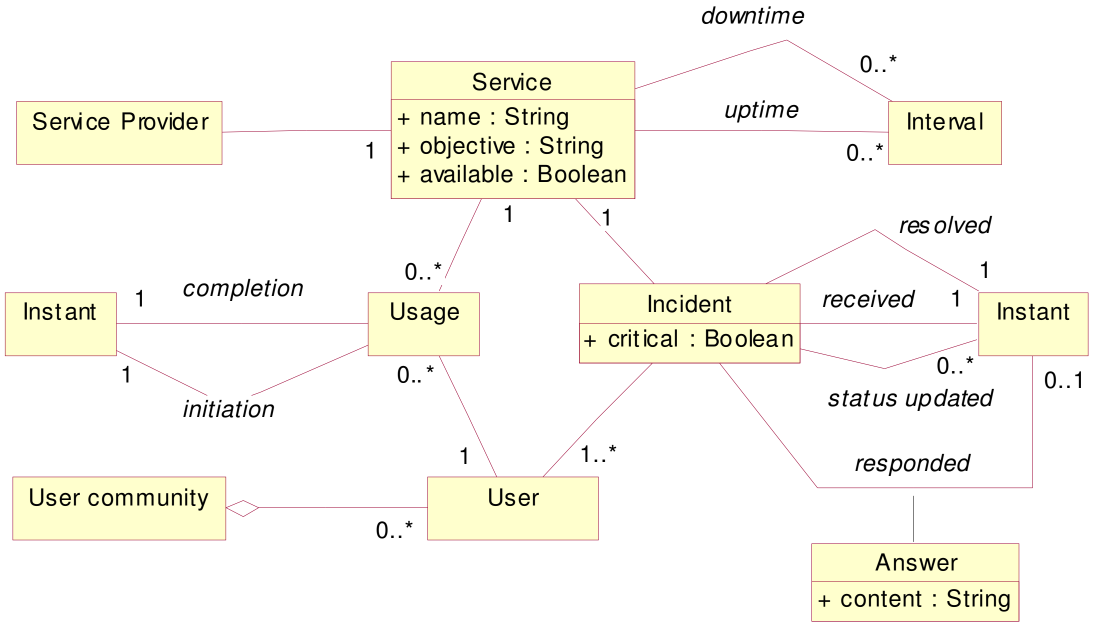


Figure 23: Service Level Agreement

Based on this model, an SLA can be specified in CQML as follows.

The ServiceProvider offers a set of services with the QoS specified in an SLA. In other words, we can specify a QoS profile to define an SLA:

```
profile SLA for ServiceProvider {
  provides   good_availability(Service) and
             fast_transactions(Service) and
             fast_multiple_users_responses(Service) and
             fast_single_user_responses(Service) and
             fast_non-critical_request_responses(Service);
}
```

This profile includes specification of availability (addressed in the first guarantee in the template) and response times (addressed in the second through fifth guarantee). Based on the first guarantee, the QoS statement `good_availabililty` can be defined as:

```
quality good_availability (s:Service) {
  availability(s) >= AVAILABILITY_PERCENTAGE and
  outage_length(s).maximum <= MAX_OUTAGE_LENGTH and
  outages(s, A_MONTH) <= MAX_MONTHLY_OUTAGES;
}
```

In this QoS statement, we have used constants in upper case letters to denote the values that need to be specified in an instance of the SLA template.

The three QoS characteristics used in the previous QoS statement have to be defined next:

```
quality_characteristic availability (s:Service) {
  domain: increasing numeric natural [0..100] percent;
  values: (100*s.uptime->sum).div(s.uptime->sum +
                                  s.downtime->sum);
}

quality_characteristic outage_length (s:Service) {
  domain: decreasing numeric natural seconds;
  values: s.downtime->last.stop - s.downtime->last.start;
}
```

Note that Interval has start and stop Instants from the model of time in subsection 4.5.2.

```
quality_characteristic outages (s:Service, duration : Integer)
{
  domain: decreasing numeric natural number_of_outages;
  values: s.downtime->select(
           start > self.downtime->last.start - duration)->size;
}
```

The specified QoS characteristics define precisely what is meant by the availability guarantee in the SLA, in terms of the model of the service provision. Next, we define the response time guarantee for the service transactions:

```
quality fast_transactions (s:Service) {
  on_time_transactions(s, MAX_RESPONSE_TIME, A_WEEK)
                                              > MIN_PERCENTAGE;
}
```

with the QoS characteristic defined as:

```
quality_characteristic on_time_transactions (
                                 s:Service;
                                 max_duration : Integer;
                                 measured_duration : Integer) {
  domain: increasing numeric natural [0..100] percent;
  values: (100*s.Usage->select(
   (completion - initiation > max_duration) and
   (s.Usage->last.initiation - initiation < measured_duration)
   )->size).div(s.Usage->select(
    s.Usage->last.initiation - initiation < measured_duration
   )->size);
}
```

The last three QoS statements in the SLA-profile all specify response times, differed only by the type of events that the service provider is to respond to.

```
quality fast_multiple_users_responses(s:Service){
  response_time(s, 2, TRUE).maximum <= MULTIPLE_RESPONSE_TIME
  and status_update_time(s, 2, TRUE).maximum <=
                       MULTIPLE_STATUS_TIME and
  resolve_time(s, 2, TRUE).maximum <= MULTIPLE_RESOLVE_TIME;
}

quality fast_single_user_responses(s:Service) {
  response_time(s, 1, TRUE).maximum <= SINGLE_RESPONSE_TIME
  and
  status_update_time(s, 1, TRUE).maximum <= SINGLE_STATUS_TIME
  and resolve_time(s, 1, TRUE).maximum <= SINGLE_RESOLVE_TIME;
}

quality fast_non-critical_request_responses(s:Service) {
  response_time(s, 1, FALSE).maximum <= REQUEST_RESPONSE_TIME
  and status_update_time(s, 1, FALSE).maximum <=
  REQUEST_STATUS_TIME and resolve_time(s, 1, FALSE).maximum <=
  REQUEST_RESOLVE_TIME;
}
```

In order to be able to define general QoS characteristics, we have introduced parameters that specify the number of users that at least must be involved in the incident and whether the incident is critical or not. We define the QoS characteristics as follows:

```
quality_characteristic response_time (
        s:Service; max_users : Integer; critical : Boolean) {
  domain: decreasing numeric seconds;
  values: if(   s.Incident->last.critical = critical and
                s.Incident->last.User->size >= max_users)
          then  s.Incident->last.responded -
                s.Incident->last.received
          else 0
          endif
}

quality_characteristic status_update_time (
        s:Service; max_users : Integer; critical : Boolean) {
  domain: decreasing numeric seconds;
  values:  if(   s.Incident->last.critical = critical and
                s.Incident->last.User->size >= max_users)
          then  s.Incident->last.status_updated->last -
                if s.Incident->last.status_updated->size = 1
                then s.Incident->last.received
                else s.Incident->last.status_updated->at(
                     s.Incident->last.status_updated->size -1)
                endif
          else 0
          endif
}
quality_characteristic resolve_time (
        s:Service; max_users : Integer; critical : Boolean) {
  domain: decreasing numeric seconds;
  values:  if(   s.Incident->last.critical = critical and
                s.Incident->last.User->size >= max_users)
          then  s.Incident->last.resolved -
                s.Incident->last.received
          else 0
          endif
}
```

This concludes the specification of the SLA using CQML.

Comparing the specifications above with the template given in Figure 22, one may claim that Figure 22 is easier to understand for humans than the UML model with the accompanying CQML specification. However, the formal model using CQML can be used in the development of the services that implement the SLA specification. Furthermore, and maybe more importantly, the CQML specification defines precisely the required measurements in order to monitor the agreement.

5.3 CQML in Process Assessment

Process assessment denotes examination of the processes used by an organisation to determine whether they are effective in achieving their goals. Process assessment is used for process capability evaluation and process improvement. In order to perform process assessment, process attributes are identified. A process attribute is a measurable characteristic of process capability. When performing process assessment, values of process attributes for the process being assessed are determined, and from these, process capability levels are derived.

The processes used by an organisation are often termed business processes. A number of definitions of the term business process exist, as for instance reported in [Marshall, 2000]. Most definitions are variations of the same, defining a business process (also known as enterprise process or simply process) as a collection of actions (also known as activities, operations or steps) taking place in a prescribed manner (also known as sequence, partial order, causality or series) and leading to the accomplishment of some result (also known as goal, purpose, outcome, output or objective). The progress of business processes is determined by business events, i.e., certain occurrences trigger activities in the process.

Both the results of a process and the process itself may have some quality aspects that are useful to model. Results of processes can be viewed as products (in a wide sense), and concepts used to model product quality as presented in subsection 2.7.1 can be used to model process results. However, even if a process delivers a high-quality result, other aspects such as stability, timeliness and effort may also influence process quality. Initiatives such as the Capability Maturity Model (CMM) [Software Engineering Institute, 2000] and ISO 9000-3 [ISO, 1991] claim that if you are concerned with ensuring the quality of a software product or service, you must concern yourself with the processes involved in producing and delivering it. The ISO 15504 model [ISO/IEC JTC1/SC7, 1998] (also known as SPICE – Software Process Improvement and Capability dEtermination) is a model for software process assessment in which a number of software processes are identified and categorised. Capabilities of these processes are expressed in terms of process attributes. The capability dimension defines a measurement scale of six capability levels that characterise any process[7]. The measure of capability is determined by measurement of process attributes on a percentage scale.

With the ISO 15504 model, CQML can be used in process assessment as follows.

- The process attributes in the ISO 15504 model can be modelled as QoS characteristics in CQML. For instance, the Process Performance attribute defined in the ISO 15504 model can be specified as:
```
quality_characteristic Process_Performance {
  domain: increasing numeric
               integer [0..100] percentage;
}
```
- In the ISO 15504 model, the process capability levels are defined as certain levels of the process attributes. For instance, for a process to be on the Managed capability level (level 2), it needs a rating of the Process Performance attribute above 85% (i.e., fully achieved) while Performance Management and Work Product Management need to have ratings above 50% (i.e., largely or fully achieved). Assuming the appropriate attribute definitions, this can be specified in CQML as:
```
quality Managed {
   Process_Performance > 85 and
   Performance_Management > 50 and
   Work_Product_Management > 50;
}
```

Using CQML for this purpose may help one to organise the process assessment, but it will in most cases not be worth the effort unless the process attributes can be automatically measured. Process monitoring is the activity of measuring and tracking the progress of a process. Such measures are updated during process execution. The characteristics to measure in process monitoring are defined in a process model during process design. If a process model is defined that generates appropriate events when being executed, QoS characteristics can be defined in CQML that the runtime environment can use when monitoring the processes. Such CQML QoS characteristics will extend the ISO 15504 process attributes with parameters denoting the modelling elements to measure and a value-specification defining how the values of the process attribute are derived.

Note that while the ISO 15504 is used to characterise the capabilities of the processes defined in an organisation, measurements of process progress can be used to denote the actual performance of the processes used in an organisation. The consistency between the defined and the executed processes is termed process conformance and is sometimes also important to characterise. In [Sørumgård, 1997], process conformance is investigated in detail. Sørumgård's approach is to define a deviation vector between the defined process and the actual observations of process execution. The deviation vector is defined along orthogonal dimensions that are part of both the defined and the measured process descriptions. The only requirement of the dimensions is that they are measured at least on an ordinal scale. Since CQML includes facilities to define QoS characteristics that can precisely define the dimensions and that are at least on an ordinal scale, the formulas in [Sørumgård, 1997] that define process conformance can be based on QoS characteristics defined in CQML.

[7] The ISO 15504 capability levels are not the same as the capability levels in CMM, but work is ongoing to harmonise the two models [Paulk, 1999].

5.4 UML Profile for QoS

UML is defined by its meta-model. In [OMG, 1999b], it is discussed how specific domains that require a specialisation of the general UML meta-model can focus UML to more precisely describe the domains by defining a UML profile. For instance, the real-time domain requires a specialisation of the UML meta-model to better suit its needs. [OMG, 2000b] is a response to the request for proposals for a UML profile for schedulability, performance, and time. Even if concrete UML profiles have started to emerge, the profiling mechanism is still discussed [D'Souza et al., 1999, Dykman et al., 1999, Clark et al., 2000, Atkinson and Kühne, 2000]. Here we choose the approach described in [OMG, 1999b] in which a UML profile identifies a subset of the UML meta-model, specifies additional well-formedness rules (expressed in OCL or in natural language) to those specified in the UML meta-model, and specifies new modelling elements with associated semantics. Using this approach, we propose a UML profile for QoS based on the concepts defined in CQML. This UML profile precisely defines the concepts needed to model QoS using UML version 1.3. We do not assume intimate knowledge about the UML meta-model; we include the parts of it that are used in the well-formedness rules in the following.

5.4.1 Orthogonality

As previously discussed and reflected in requirement 8 in section 4.3, we believe the QoS-aspects of an entity should be treated as orthogonal to its functional aspects. This view is reflected in this QoS profile by defining a separate package in the meta-model of QoS concepts that refer to concepts in the standard UML meta-model. Figure 24 reflects this; the QoS concepts in the Quality Elements package refer to the standard UML concepts in the standard Foundation package, but not the other way around.

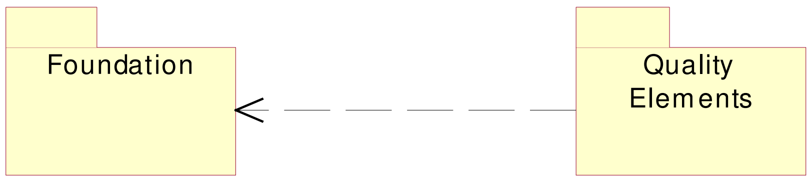


Figure 24: QoS concepts are orthogonal to standard UML concepts

5.4.2 Conceptual Model

A conceptual model, sometimes referred to as a virtual meta-model, describes the fundamental abstractions related to the domain of the profile. A conceptual model is independent of the UML meta-model, but it defines the concepts that are to be related to the standard UML elements and hence, is often included as an informative part of a UML profile. For most purposes, the concept model is of more interest than a representation in the UML meta-model of the concepts therein. For instance, whereas a QoS statement in CQML essentially is an association between a QoS characteristic and a value, it would in the UML meta-model typically be represented as a special kind of association (subtype of meta-class Association) with appropriate well-formedness rules (e.g., constraining the types of the AssociationEnds). Tool builders appreciate the latter kind of information while profile users usually have more interest in the conceptual model. Moreover, given a well-defined conceptual model, unique specialisations of modelling elements in the UML meta-model can be defined relatively easily; this part of a UML profile is mostly a matter of choosing one specialisation that all tool builders must follow. Therefore, we do not specify UML meta-class specialisations of the complete set of concepts represented in the conceptual model, we only introduce such specialisation wherever they are conceptually important.

In this QoS profile, the conceptual model is a model of the concepts defined in CQML. The model in Figure 25 is a simplification of the model presented in Chapter 4 and shows the overall structure of the CQML concepts that is detailed further in subsection 5.4.3.

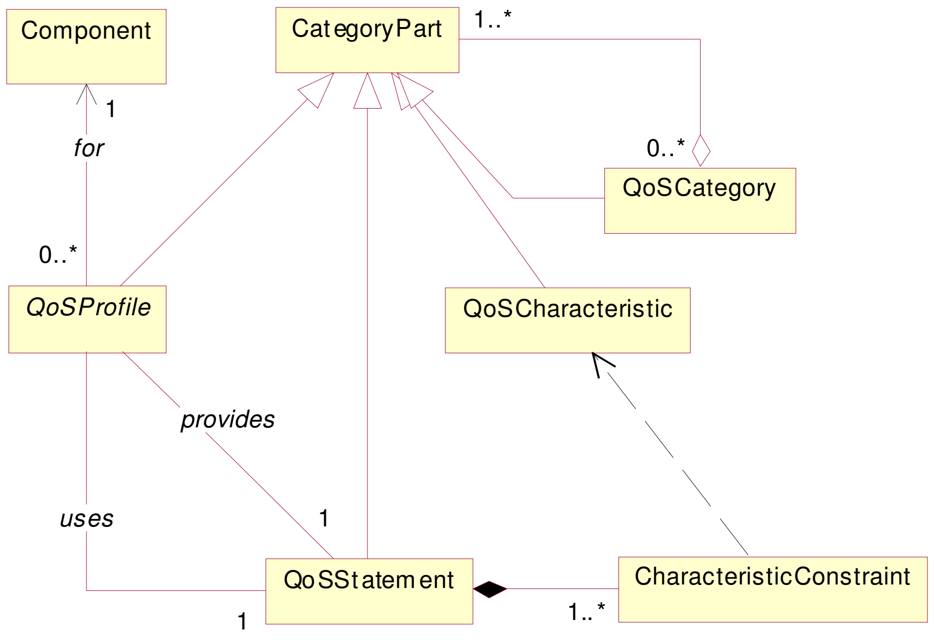


Figure 25: QoS conceptual model

5.4.3 Profile

A profile in CQML (to be distinguished from the UML QoS profile presented in this subsection) relates specifications of QoS to other modelling elements. Figure 26 shows a conceptual model of the CQML QoS profile (termed QoSProfile in the model) and shows how QoSProfile can be related to elements in the UML meta-model.

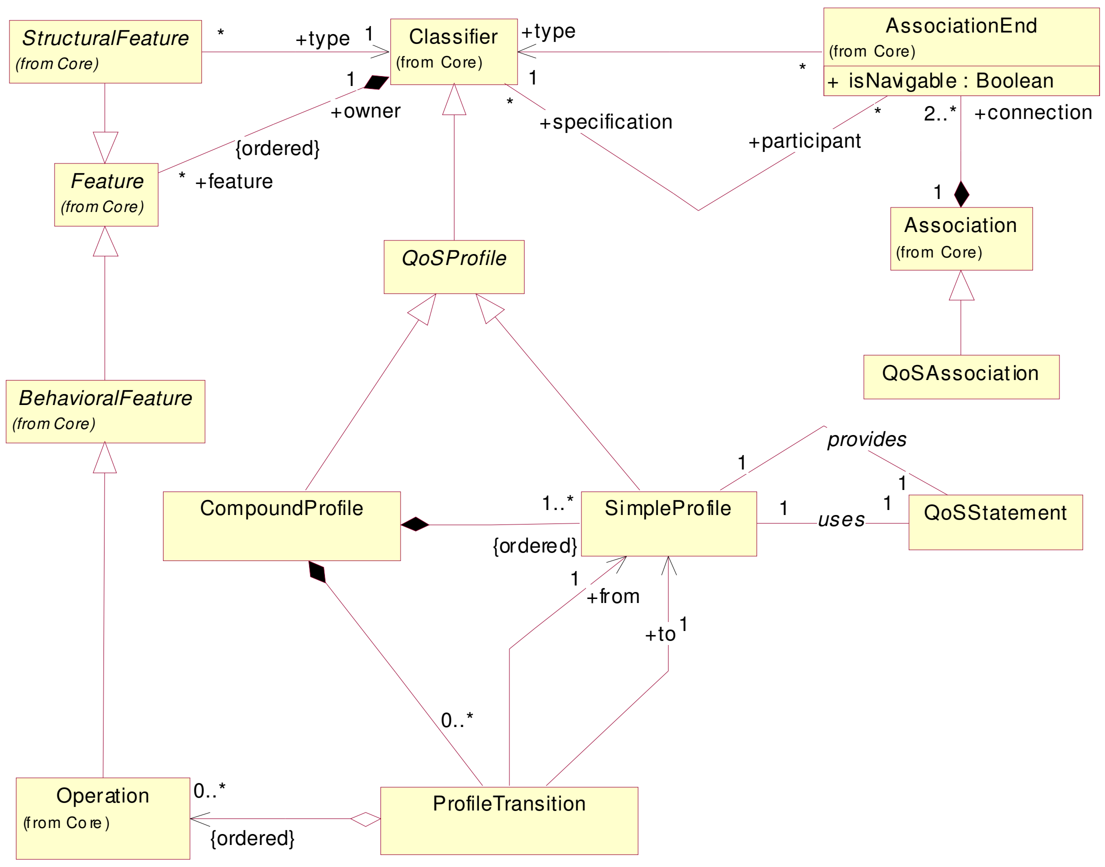


Figure 26: Conceptual model of QoS profile

The model shows that QoSProfile is a Classifier, i.e., that it defines a Namespace and that it is a GeneralizableElement (so it can be refined). It also shows a QoSAssociation that is a special kind of Association. This is used to represent the fact that a QoS profile in CQML is defined for one type of component, i.e., the *for*-association between QoSProfile and Component in Figure 25. Note that a Component in UML denotes a physical piece of implementation of a system and does not fit the notion of Component used in CQML and that is used in Figure 25. Instead, we use Classifier for the target of the QoSAssociation.

5.4.3.1 Well-formedness Rules

[1] A QoSAssociation is between a QoS profile and a Classifier.

```
context QoSAssociation inv:
-- The association is between exactly two instances
connection->size = 2 and
-- The first AssociationEnd is a Classifier
-- (or a subtype)
not (connection->first.type.oclAsType(Classifier) =
     Undefined) and
-- The second AssociationEnd is a QoSProfile
-- (or a subtype)
not (connection->last.type.oclAsType(QoSProfile) =
     Undefined)
```

[2] A Classifier cannot navigate to its QoSProfile(s), preserving orthogonality.

```
context QoSAssociation inv:
-- The Classifier cannot navigate to the Profile
not connection->last.isNavigable
```

[3] A QoSProfile is defined for one Classifier, i.e., it must have exactly one QoSAssociation.

```
context QoSProfile inv:
-- QoSProfile participates in exactly one QoSAssociation
self.participant->select(a | a.Association
   .oclIsKindOf(QoSAssociation) )->size = 1
```

[4] Transitional behaviour is defined by operations on the Classifier that the profile is for.

```
context CompoundProfile inv:
ProfileTransition->forAll(t | t.Operation->forAll(
      op | op.owner = self.participant->select(a |
            a.Association.oclIsKindOf(QoSAssociation))
                  .connection->first.type ))
```

Note that the last well-formedness rule traverses associations (the aggregates between CQML concepts such as between ProfileTransition and Operation) on the conceptual model that in a complete UML profile would need to be specified as specific kinds of Association.

5.4.4 Statement

A CQML quality profile specifies QoS offers and QoS expectations as QoS statements. Figure 27 shows the conceptual model of QoS statement.

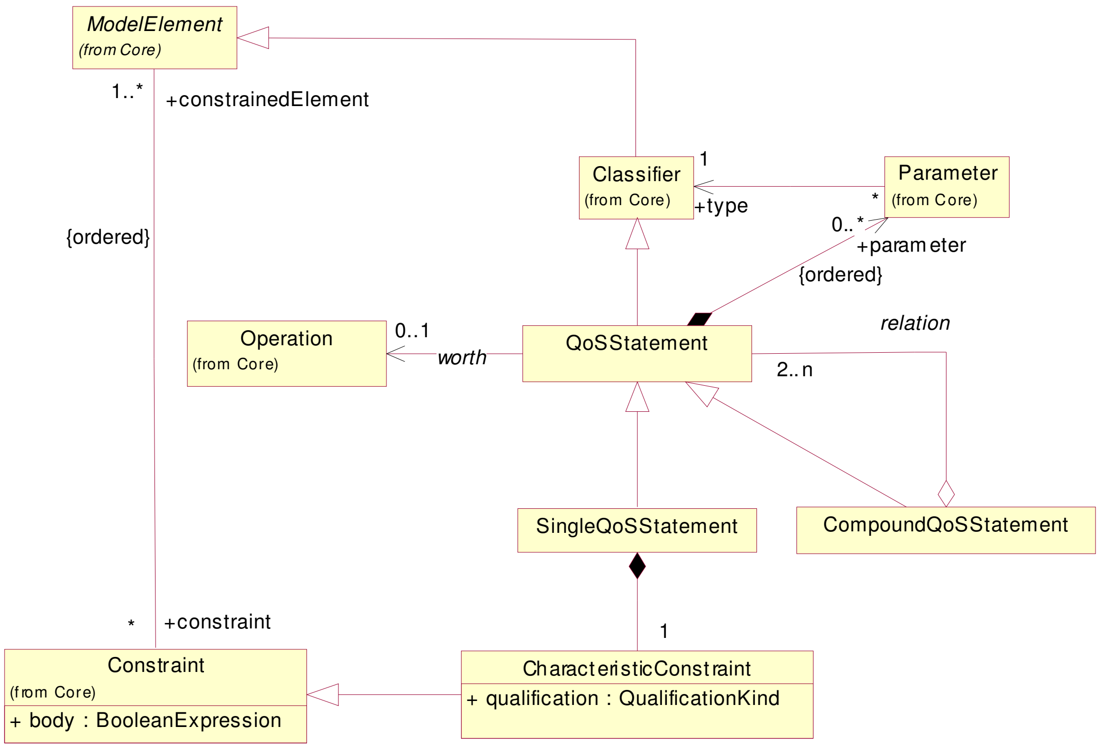


Figure 27: Conceptual model of QoS statement

The model shows that QoSStatement is a Classifier. It also shows that QoSStatement may be single or compound. A compound QoS statement contains two or more QoS statements, and relates these by the *relation*-association. This models the fact that a compound QoS statement relates other QoS statements by *and*- or *or*-relations. Furthermore, the model shows that a SingleQoSStatement contains a CharacteristicConstraint. The CharacteristicConstraint is a Constraint and contains an attribute that specifies the qualification of the constraint (i.e., best-effort, guaranteed, etc).

5.4.4.1 Well-formedness Rules

[1] The constraint comprising a SingleQoSStatement restricts a single QoS characteristic.

```
context CharacteristicConstraint inv:
-- It constrains only one element
constrainedElement->size = 1 and
-- The constrained element is a QoSCharacteristic
constrainedElement.oclIsKindOf(QoSCharacteristic)
```

The actual constraint is represented in a BooleanExpression in the *body*-attribute of Constraint.

[2] The worth function must be an operation of the Classifier.

```
context QoSStatement inv:
-- Either is the worth-function not specified
worth = Undefined or
-- or the operation is of the classifier
self.SimpleProfile.participant->forAll(a |
   a.Association.oclIsKindOf(QoSAssociation) and
     a.Association.connection->first.type = worth.owner)
```

[3] The parameters of the QoSStatement must be StructuralFeatures of the Classifier or its parts.

```
context QoSStatement inv:
let classifier = self.SimpleProfile.participant->select(
           a | a.Association.oclIsKindOf(QoSAssociation))
                   .connection->first.type in
     classifier.allStructuralFeatureTypes()->
     includesAll(parameter.type)
```

In this rule we use a non-standard operation on classifier that can be defined as:

```
context Classifier::allStructuralFeatureTypes() :
Set(Classifier)
post:
    self.feature->iterate(
       f : Feature;
       result : Set(Classifier) = Set{self} |
       if f.oclIsKindOf(StructuralFeature) then
         result = result->union(
           f.type.allStructuralFeatureTypes())
       else result
       endif)
```

The association from QoSStatement to SimpleProfile that is used in the last two well-formedness rules is a shorthand for either one of the `uses` or the `provides` associations in Figure 26. The properties of Operation that are used in the last two well-formedness rules are only shown in Figure 26.

5.4.5 Characteristic

A CQML QoS characteristic defines a dimension in which QoS can be specified and measured. Figure 28 shows the conceptual model of QoS characteristic.

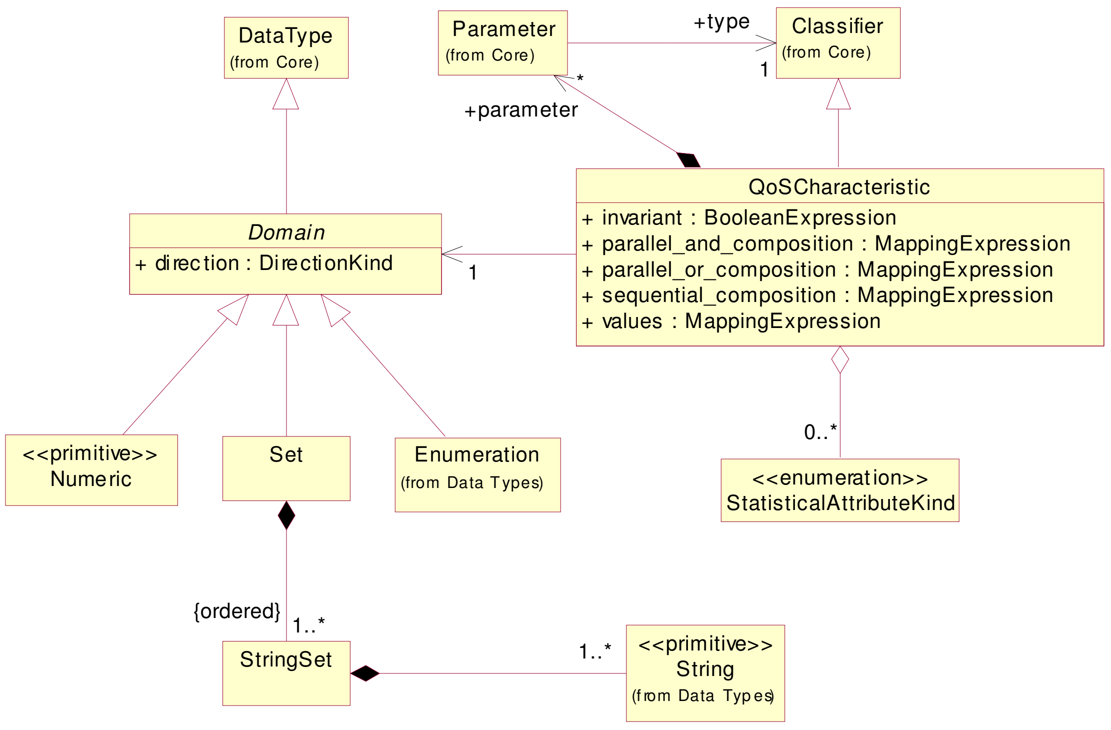


Figure 28: Conceptual model of QoS characteristic

The model shows that QoSCharacteristic is a Classifier and that it contains attributes for the different kind of expressions that can be specified as part of a QoS characteristic. It also shows that a QoSCharacteristic has a Domain with a direction. The three composition attributes and the values attribute are MappingExpressions, i.e., expressions that each evaluates to a mapping between modelling elements. In the UML semantics, the format of such mappings is not specified and, hence, we use only natural language to specify well-formedness rules for these expressions. The same is true for BooleanExpression that we use for the invariant attribute.

5.4.5.1 Well-formedness Rules

[1] The parameters of a QoSCharacteristic must be the Classifiers (or StructuralFeatures of them) that are referred to in the parameters in QoSStatements that constrain the QoSCharacteristic.

```
context QoSCharacteristic inv:
let statements = self.constraint->iterate(
      c : Constraint;
      result : Set(Classifier) = Set{} |
      if c.oclIsKindOf(CharacteristicConstraint) then
        result = result->union(c.SimpleQoSStatement)))
      else result
      endif in
statements->forAll(s | s.allStructuralFeatureTypes()
   ->includesAll(self.parameter.type))
```

The associations from QoSCharacteristic (as a ModelElement) to CharacteristicConstraint and from CharacteristicConstraint to SimpleQoSStatement that are used here are shown previously and not repeated in Figure 28.

[2] The three composition-expressions in QoSCharacteristic map two values of Domain into a new value of Domain.

[3] The values-expression in QoSCharacteristic maps values of the Features of the parameters into a value of Domain.

[4] The invariant-expression may only refer to Features of the parameters, values of the Domain and the StatisticalAttributeKinds.

[5] The parallel_and composition is used when the two values are parts of an aggregate.

[6] The sequential composition is used when the two values are parts of a composite.

[7] The parallel_or composition is used when the two values are parts of an aggregate and an or-constraint exists between the two aggregate associations.

5.4.6 Category

As illustrated in Figure 29, the CQML QoS category is a namespace for QoS profile, QoS statement and QoS characteristic.

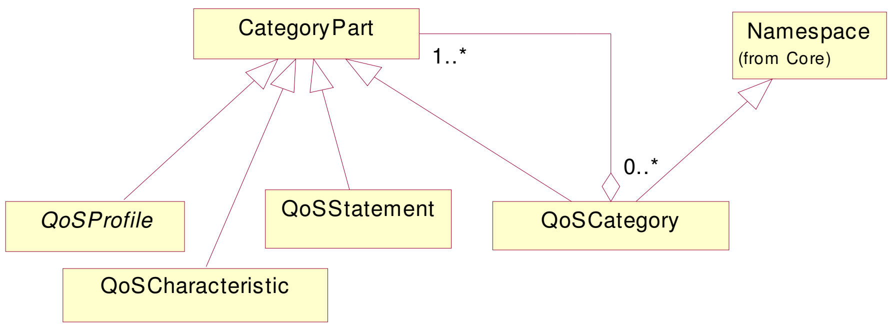


Figure 29: Conceptual model of QoS category

5.4.6.1 Well-formedness Rules

None.

5.4.7 Quality types

As illustrated in Figure 30, the data types used in CQML are special kind of DataType.

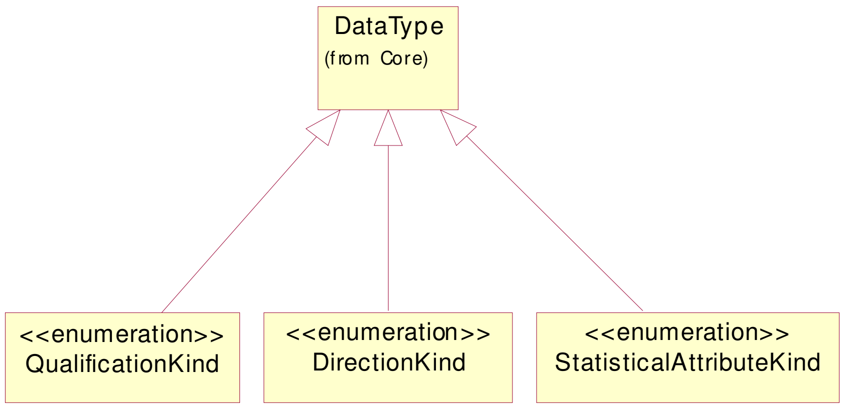


Figure 30: Conceptual meta-model of quality types

QualificationKind defines an enumeration whose values are *guaranteed*, *best-effort*, *threshold best-effort*, *compulsory best-effort*.

DirectionKind defines an enumeration whose values are *Undefined*, *increasing*, *decreasing*.

StatisticalAttributeKind defines an enumeration whose values are *maximum*, *minimum*, *range*, *mean*, *variance*, *standard_deviation*, *percentile*, *frequency*, *moment*, *distribution*.

5.4.7.1 Well-formedness Rules

None.

5.4.8 Relation to UML Meta-model

As already pointed out, a UML profile identifies a subset of the UML modelling concepts, provides additional well-formedness rules and defines new modelling concepts. Such new modelling concepts may be specified in conceptual models, but, as discussed in subsection 5.4.2, to relate them to the UML meta-model, corresponding UML stereotypes need to be defined, i.e., subclasses of existing classes in the UML meta-model. The conceptual models shown in the previous subsection introduce new modelling elements as subclasses of UML meta-classes directly, without defining them as subclasses of Stereotype with an appropriate value of the attribute baseClass. Furthermore, to formally specify a UML profile, all associations between new modelling elements in the conceptual meta-models need to be defined as stereotypes of existing meta-classes (e.g., Association). Finally, some of the new modelling concepts have attributes while stereotypes can only be defined as restrictions on existing meta-classes. The attributes introduced in the conceptual models above must therefore be modelled as tagged values on the stereotypes (sometimes referred to as pseudoattributes).

5.4.8.1 Notation

Since all new modelling concepts above are defined as stereotypes, they can be used in models as ordinary stereotypes (using guillemets). Standard QoS concepts (such as delay, throughput, and jitter) can be defined using these stereotypes and used as tags on ordinary modelling elements. In section 5.5, we show in a case study an example of how this can be used.

5.4.9 Tool Chain

To arrange for the inclusion of QoS in software development, a tool chain that takes QoS concepts into account is needed. Such a tool chain should include a modelling tool and a tool that transforms the models to an appropriate format for the run-time environment when it performs QoS management. Such a tool chain can be implemented as follows.

The UML QoS profile defined here needs tool support to become practical to use for software developers. Such tool support should be integrated with the modelling tool suite used since the new concepts in the UML QoS profile strongly relate to existing UML modelling elements. Such integration can be achieved by using tools, such as Objecteering [Softeam, 2000], that allow extensions to their representation of the UML meta-model, i.e., tools that allow definition of stereotypes that are then treated as first-class modelling elements. Since the UML QoS profile is based on CQML, it is straightforward to transform a UML representation of the CQML concepts in the modelling tool to the lexical CQML, to enable the CQML compiler to be used. The compiler parses CQML and fills a repository for the run-time environment to access when its performs QoS management, as discussed in Chapter 6.

5.5 CQML in OO-development

Using the QoS profile defined in the previous section, we show here how CQML can be included in object-oriented software development.

5.5.1 Introduction

Object-oriented modelling of distributed systems has been a focus at SINTEF for a number of years. Results from this work have been reported in [Berre and Aagedal, 1996, Aagedal et al., 1997b, Oldevik and Berre, 1998, Silva et al., 1997, Berre et al., 1997, Solberg et al., 1999], and in project reports from the European ESPRIT-IV projects OBOE and DISGIS. In addition to ongoing work in our group for distributed information systems at SINTEF and with our clients, discussions with Trygve Reenskaug, Alistair Cockburn, Earl Ecklund and others have resulted in valuable insights our group has used in the continuous refinement of a distributed systems development methodology.

We do not believe there is such a thing as a “right” software development methodology. Different types of organisations, different software developers and different types of software may all have different requirements towards a software methodology. A seemingly countless number of life-cycle models and software methodologies have been postulated over the years, all claiming to have advantages that others do not have. Some of these claims are supported by case studies, showing how a particular methodology can be successfully applied in a particular context. However, given the number of uncontrollable factors, experiments will never give us the answer as to what methodology is “right” even if such a thing were to exist. Nevertheless, by specifying desirable characteristics and reasoning about their support, we can argue that some methodologies are more suitable to be used for certain tasks than others are. In this thesis, we rely on work on general software development methodologies and add support to model QoS. We cannot claim a methodology resulting from using our QoS additions is “right”, but we argue we make QoS aspects visible when doing modelling in such a way that conscious decisions on QoS can be made during software development. We believe that having a conscious view on QoS during development will benefit the total costs of developing QoS-aware systems.

In a specific software development project, the methodology that is used needs to be tailored to the specific requirements of the project and to the specific personnel and organisation. We do not focus on alternative routes through a methodology, neither do we give guidelines for when and how to decide on such routes. We present the Lecture-on-Demand (LoD) case study in a straightforward sequence of business modelling, system modelling, design and implementation. This does not imply that we prescribe this sequence, modelling activities may be performed either in a depth-first manner, breadth-first manner, or both (i.e., following an iterative and incremental approach as prescribed for instance in the Unified Process [Jacobson et al., 1999]).

In the paper included in appendix I.C, we present an approach to consider QoS aspects in system development using UML and an ODP-based approach. The case study presented next builds on this by extending the approach to use the UML profile defined previously and by including CQML as a precise language to specify QoS.

5.5.2 LoD Case Study

Distance education, sometimes more fashionably termed e-learning, has received considerable attention both commercially and in the research community. The increasing capabilities of networking and multimedia technologies have contributed to the gain of interest. LoD is one kind of distance education where students can attend lectures over the internet at times of their choice. We have performed a case study to design such a system and report in the following parts of this design in which QoS pertain. We focus on those parts of the LoD design that highlight how QoS can be modelled at different levels of abstraction.

5.5.3 LoD Business Modelling

5.5.3.1 Introduction

The objective of business modelling (often referred to as enterprise modelling) is to get a formal representation of essential aspects of an enterprise, so that one can make well-founded decisions on what computer system support one needs to improve some aspects of the business. Actually, one may sometimes conclude that one does not need computer system support; only organisational changes may be required. However, the decision to initiate business modelling usually comes from an acknowledgement that the way one performs business may be improved by introducing computerised support. Hence, business modelling is often a first step in a computer system development process.

[Aagedal and Milosevic, 1999] and [Aagedal and Milosevic, 1998] (the papers presented in appendices I.A and I.B) focus on enterprise modelling. In [Aagedal and Milosevic, 1999] the suitability of UML for enterprise modelling is discussed. Even if we identified parts of the RM-ODP enterprise language, at that time, that UML was unsuited to specify, we are not aware of alternative modelling languages that fully address the RM-ODP enterprise language. In particular, policy specifications that include deontic concepts are difficult to specify in UML or any other language. We therefore use UML for business modelling, adhering to the usage specified in [Aagedal and Milosevic, 1999]. The same is true for specification of QoS in enterprise modelling, as is the topic of [Aagedal and Milosevic, 1998].

5.5.3.2 Business Use Case Model

The LoD business use case model included below shows two types of LoD-viewers, their business use cases and appropriate constraints. A business use case represents an activity some business unit is responsible to perform. A business unit is an identifiable business entity that may be or consist of a computer system, and it corresponds to a community in RM-ODP. Constraints on the dependencies between actors and use cases correspond to contracts between communities.

A Viewer may view a lecture, but only a Student may ask questions and get answers. View Lecture is the main use case. An important quality constraint on this use case is that learnability should be good. Learnability of a lecture viewed using the LoD system depends on a number of factors such as the media quality of the lecture viewed, its pedagogic quality, usability of the system, etc. Hence, learnability depends both on the capabilities of the LoD system and on other factors such as the information used by the LoD system when showing a lecture. However, in a business use case model, both human actors and computerised systems may perform the use cases. Hence, the quality constraints pertaining to the LoD system will not be made clear until the LoD system is identified as a separate entity in system modelling.

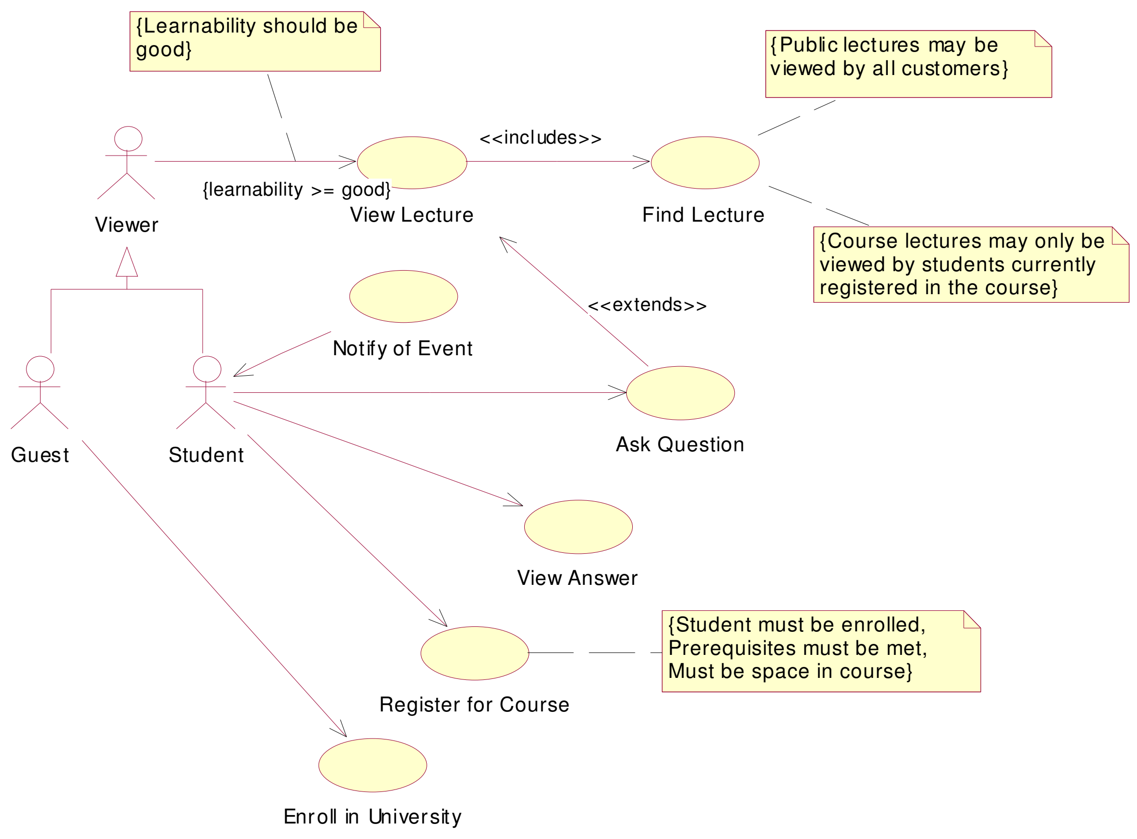

{Learnability should be good}
{Public lectures may be viewed by all customers}
<<includes>>
Viewer
{learnability >= good}
View Lecture
Find Lecture
{Course lectures may only be viewed by students currently registered in the course}
<<extends>>
Notify of Event
Guest
Student
Ask Question
View Answer
{Student must be enrolled, Prerequisites must be met, Must be space in course}
Register for Course
Enroll in University

Figure 31: LoD business use case model

In [Ecklund et al., 2001b], the requirements model for the LoD case study is presented. The model presented therein elaborates the use case model in Figure 31 to include more actors and use cases.

One can already at this level introduce specifications of qualities using CQML. For instance, from the informal description in the previous subsection, learnability can be specified as:

```
quality_characteristic simple_learnability {
   domain : increasing enum{poor, average, good, excellent};
}
```

The constraint in the business use case model can then be specified as:

```
simple_learnability >= good
```

In the informal description above, it is mentioned that learnability depends on other qualities such as media quality, pedagogic quality, and usability. This fact can be specified in CQML by extending the definition of simple_learnability as follows:

```
quality_characteristic learnability : simple_learnability {
   media_quality;
   pedagogic_quality;
   usability;
}
```

In CQML, one can use the values-clause to specify how learnability depends on these other three quality characteristics and how values of each of the quality characteristics aggregate into the enumerated domain of learnability. The specification above, however, only states that such dependencies exist.

Admittedly, the natural language constraint in Figure 31 combined with the informal description of the dependencies may convey the meaning of the constraint equally well as the CQML specifications above, but to be able to trace requirements to design choices, it is useful to specify the constraints using CQML.

In a UML diagram, the learnability constraint can be specified as depicted in Figure 32.

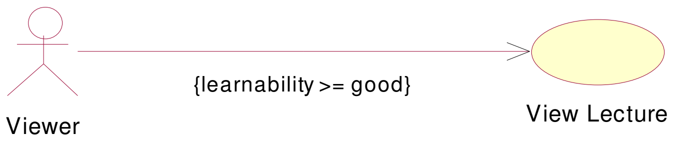


Figure 32: Learnability constraint

The constraint is specified on the association between the Viewer actor and the View Lecture use case, and the QoS characteristic used is defined in a separate classifier stereotyped as QoSCharacteristic, as shown in Figure 33.

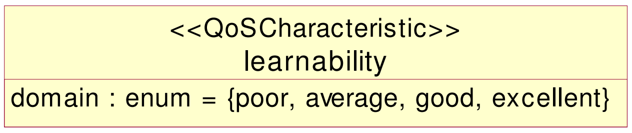


Figure 33: Learnability characteristic

The specification of a constraint on the association between an actor and a use case (as used in Figure 32) is shorthand for a profile for the use case that has a QoS offer as the quality specified in the constraint.

5.5.3.3 Business Process Model

A business process model shows the business activities and their dependencies. In addition, it may show the roles responsible for performing the activities. The LoD business process model in Figure 34 shows the process of producing multimedia (MM) lectures. The Lecturer gives the lecture while the multimedia lecture producer produces multimedia elements and collects such elements to create a multimedia lecture. In the process of giving a lecture, the lecture content is produced and used as input to the activity of producing multimedia elements. We include a constraint on the lecture content that it should be pedagogic. Even if we do not detail this constraint further, we have identified where in the system the pedagogic nature of the lecture material pertains, and that the multimedia lecture producer expects data from pedagogic lectures in order to produce multimedia lecture that has good learnability.

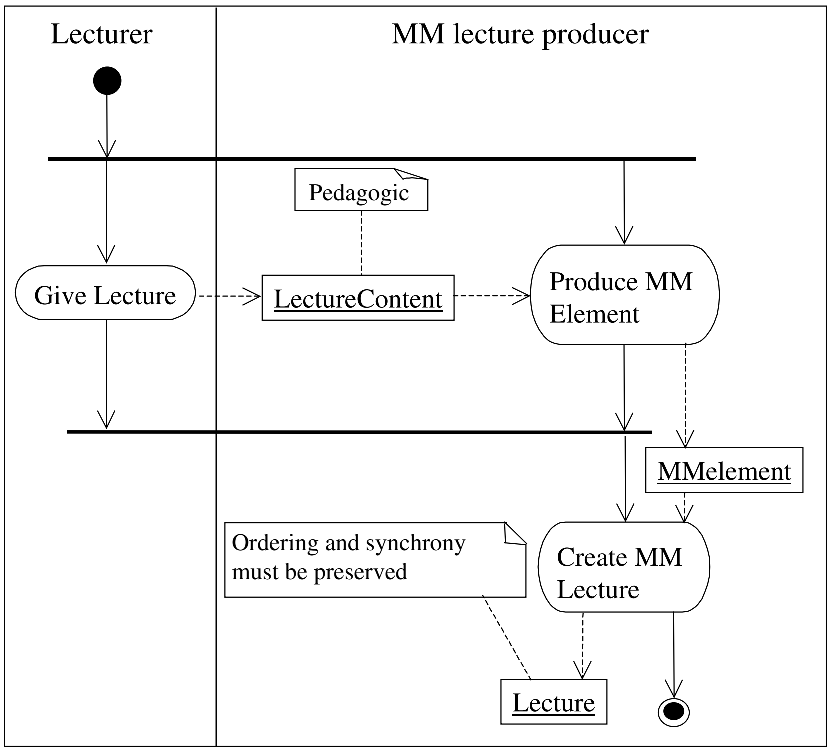


Figure 34: LoD business process model

The pedagogic quality that is required from multimedia data, also referred to in the business use case model, can be defined in CQML as:

```
quality_characteristic pedagogic_quality {
   domain : increasing enum {non-pedagogic, pedagogic};
}
```

Using this, a quality level named instructive can be defined as:

```
quality instructive {
   pedagogic_quality = pedagogic;
}
```

Then a profile for the multimedia data can be defined as:

```
profile Pedagogic for LectureContent {
   provides instructive;
}
```

Using the UML QoS profile, the constraint on the multimedia data is shown in Figure 35; the instructive quality statement is defined in Figure 36.

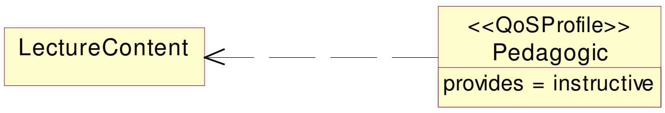


Figure 35: Pedagogic multimedia data

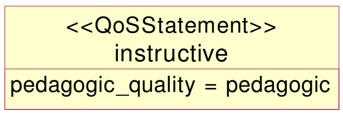


Figure 36: Instructive quality level

A multimedia element contains, in addition to the lecture content, meta-information about the lecture such as time, place, lecturer and subject, and also inter-element information such as ordering and synchrony information. We have specified a constraint on the created multimedia lecture specifying that the multimedia elements must preserve their synchrony and ordering. Presenting lecture elements in the wrong order or showing them out of synch clearly reduces learnability and should be avoided. We combine these aspects into a QoS characteristic "smoothness" defined and used below.

In CQML, the quality profile for the multimedia lecture that is the result of the final activity in the process model in Figure 34 can be defined as:

```
profile Ordered_in_Synchrony for Lecture {
   provides smooth;
}
```

The quality statement "smooth" is defined as:

```
quality smooth {
   smoothness = good;
}
```

The quality characteristic "smoothness" is defined as:

```
quality_characteristic smoothness {
   domain : increasing enum {poor, good};
}
```

The models using the UML QoS profile are straightforward and not included.

The quality constraints in the process model relate to the results of the activities rather than the activities themselves. If the quality of the activities was of interest, as when performing process assessment, the activities need to be treated as entities in their own right and appropriate quality constraints can be attached.

5.5.3.4 *Business Object Model*

In the business use case model, a number of actors are identified, and in the business process model, a number of objects produced in the business activities are identified. These are combined in the business object model, together with additional relevant business objects. Focussing on viewing lectures, the business object model in Figure 37 shows the business objects and their associations. A Viewer may view a number of lectures that each contains a number of multimedia elements that in turn contain lecture content.

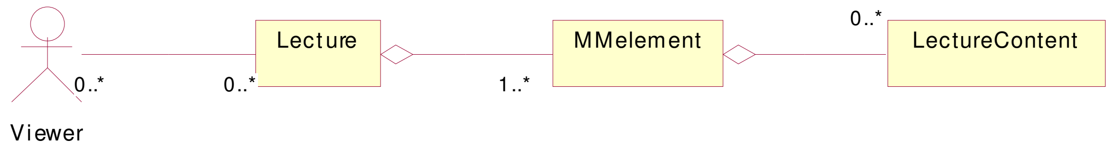


Figure 37: LoD business object model

Both LectureContent and Lecture already have QoS constraints defined from the business process model. Furthermore, the View lecture use case in the business use case model has a QoS constraint that essentially is a constraint on the association between the Viewer and Lecture. We therefore do not need to add QoS specifications on this model, but note that such specifications can be added on business distribution models.

5.5.3.5 Business Distribution Model

The business distribution model shows distribution inherent in the business. In the LoD business case, two different locations can be identified during business modelling, the Study and the University, as shown in Figure 38.

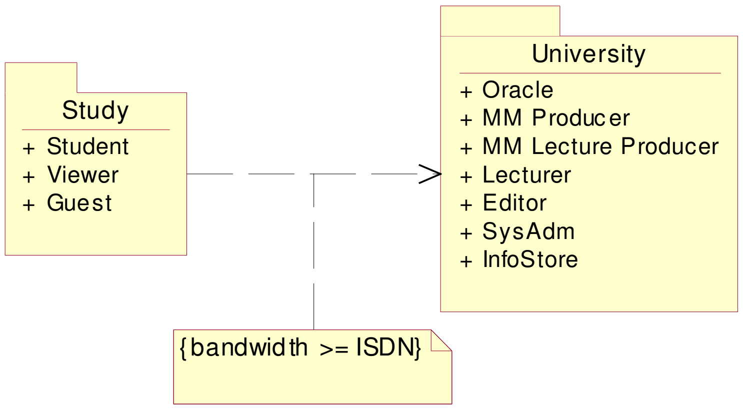


Figure 38: Business distribution model

The listed names in each location denote roles that are needed to perform business activities assigned to the distribution unit. The roles are identified as part of business process modelling or in collaboration models resulting from a separate role-modelling activity. A business distribution model highlights inherent distributed communication by showing dependencies between units of distribution. Figure 38 shows distributed communication is required for communication between the Viewer and the InfoStore, i.e., when viewing a lecture. The QoS constraint from the business use case model pertains to the communication between Study and University. In addition, a business constraint is that one cannot require Viewers to be connected with more bandwidth available than what ISDN provides (i.e., the minimum bandwidth is ISDN). This constraint will limit design choices for the communication system.

5.5.4 LoD System Modelling

During system modelling, we identify those parts of the business model the system is responsible to provide. Based on the business model, system use cases are identified and system types defined.

5.5.4.1 System Use Case Model

We combine the business use case model from Figure 31 with the business process model from Figure 34 to form a starting point for a system use case model.

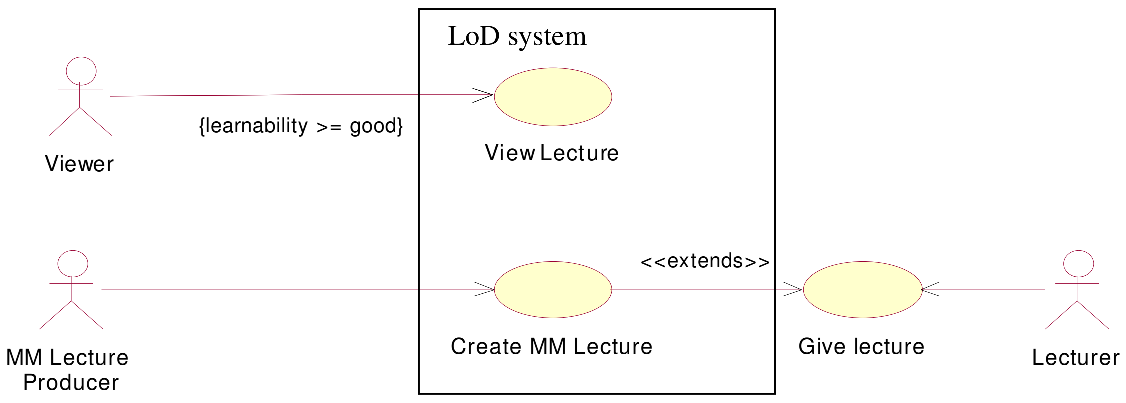


Figure 39: System use cases

The system use case model in Figure 39 shows the system boundary by specifying two use cases the LoD system is responsible to provide and one use case that is outside the LoD system. The QoS constraints pertaining to the use cases that the system is responsible for are constraints the system must adhere to, for instance, that learnability should be good when viewing a lecture.

In the initial system use case model, the system boundary is defined on the business use case model. As part of system modelling, this initial system use case model is refined and structured into more detailed models. The business distribution model may be superimposed on these models to reflect locations of use cases.

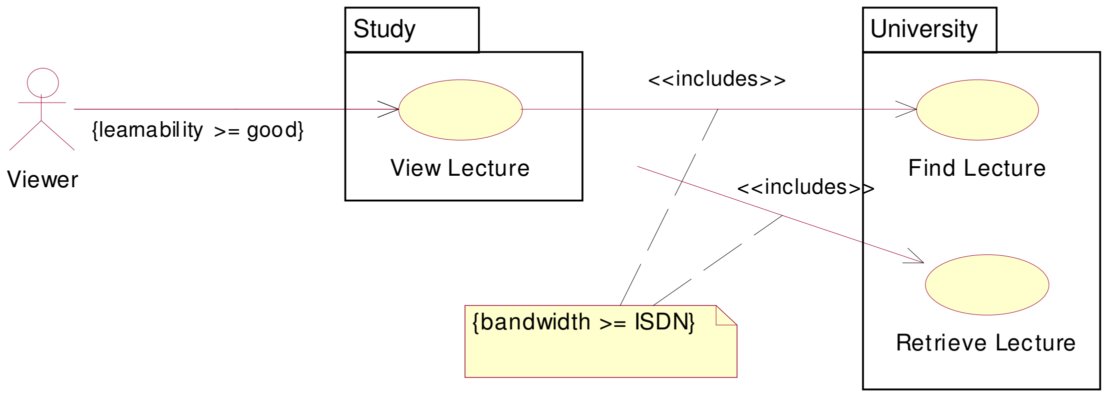


Figure 40: Structured system use case model with distribution

In Figure 40, a structured system use case model is shown where the business constraint on communication between the Study and the University locations are pinpointed to the associations between the View Lecture use case and the Find Lecture and Retrieve Lecture use cases it includes. The QoS constraint on View Lecture that learnability should be good has also consequences for the two use cases in the University since they both are included in the View Lecture use case. This means that the learnability constraint on the View Lecture use case must later be decomposed onto the parts of the system implementing the involved use cases.

5.5.4.2 System Type Model

The system type model defines externally visible system types in order to define the externally visible behaviour of the system. The externally visible system types typically originate from input and output parameters of operations the system must provide in order to implement the system use cases. The system type model is often used to precisely define behaviour by specifying pre- and postconditions on system operations using the system types as vocabulary.

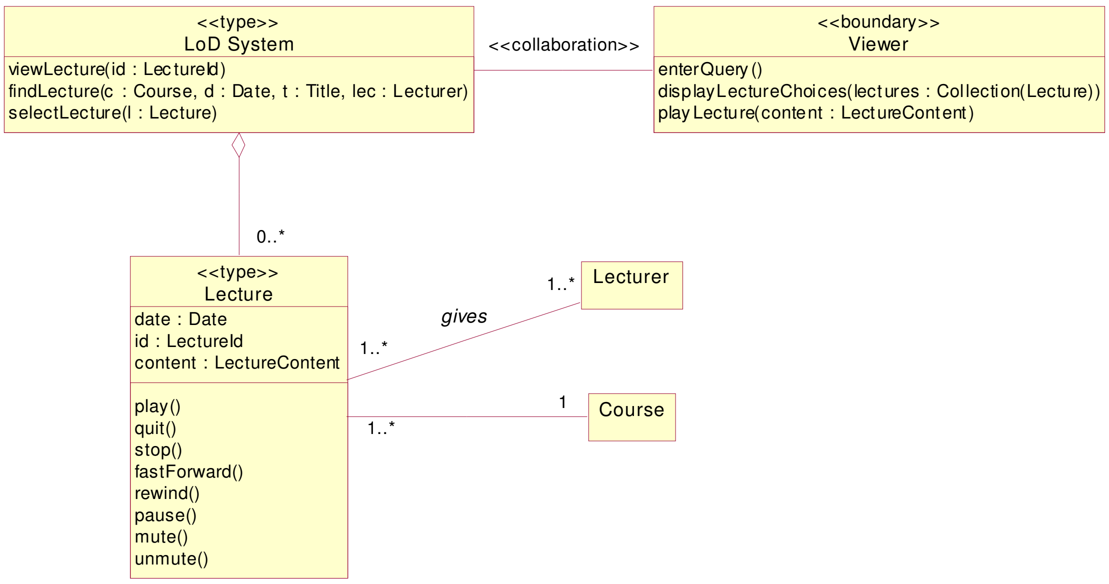


Figure 41: System type model

Figure 41 shows a system type model of the LoD system where the system types involved in viewing a lecture are included. The LoD system type is a model of the parts of the LoD system involved in collaboration with a Viewer in order to view a lecture. The model shows that the LoD system contains a number of Lectures that are related to Lecturers and Courses. The model also shows the Viewer type representing the Viewer actor. This type represents a user interface with appropriate callback operations and is therefore stereotyped as <<boundary>>.

We are now in a position to specify quality constraints pertaining to the LoD system by defining a QoS profile for it using the system types. The QoS profile in the business model illustrated in Figure 32 uses the learnability QoS characteristic, which depends on usability, pedagogic quality and media quality. The business process model in Figure 34 shows that the pedagogic quality pertains to the lecture content while the media quality pertains to the lecture. Based on this, we refine the QoS profile used in the business model in Figure 32 to:

```
profile StudentQuality for LoD_system {
   uses instructive(Lecture.content);
   provides instructive(Lecture.content) and
          usable(viewLecture, Viewer.playLecture) and
          good_media(Lecture);
}
```

This QoS profile specifies that the LoD system provides good learnability if the pedagogic quality of the lecture content is instructive (as defined in the business model). We assume the LoD system does not depreciate the pedagogic quality of the lecture content, and focus therefore on usability and media quality.

We define the usability of the system to be related to the response time from when a Viewer invokes the viewLecture operation to when the effects of the operation is visible, i.e., when the playLecture operation is invoked on the Viewer. Usability, in subsubsection 2.7.1.1 referred to as quality in use, can include many more aspects with respect to effectiveness, productivity, safety, and satisfaction than response time only, but we restrict ourselves in the following to response time. We specify a usable system to have a response time less than 1 second:

```
quality usable(cause : Operation, effect : Operation) {
   response_time(cause, effect) < 1000;
}
```

```
quality_characteristic response_time( cause : Operation,
                                      effect : Operation) {
   domain : decreasing numeric milliseconds;
   values: cause.SR->last.time() - effect.SE->time();
}
```

In addition to good quality of the individual media of the lecture, media quality also includes specification of smoothness.

```
quality good_media(lecture : Lecture) {
   smooth(lecture.content) and
   media_quality(lecture.content) >= good;
}
```

The QoS statement smooth is similar to its use in 5.5.3.3, except that we have introduced a parameter denoting the part of the system the QoS constraint pertains.

5.5.5 LoD Design

In design, we refine the system models by decomposing the system types into collaborating components fulfilling the responsibilities of their corresponding system types. We continue to focus on the View Lecture use case to illustrate the refinement of the QoS specifications.

The first step of design is often to create a high-level design model based on concepts from the business model and structured according to design decisions.

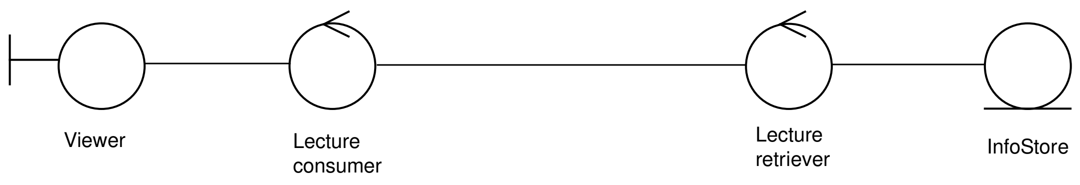


Figure 42: High-level design model

The high-level design model of viewing a lecture in Figure 42 shows the collaborating components stereotyped using the Unified Process standard analysis[8] stereotypes (boundary, control, entity). The Viewer is from the system model, while the LoD system is decomposed into Lecture consumer, Lecture retriever and InfoStore. The response time constraint from the system model can be related to this high-level design model by using a sequence diagram as shown in Figure 43.

[8] The analysis stereotypes are used when modelling internal components of the system (albeit representing concepts from the problem-domain). In our terminology, these analysis stereotypes are therefore used in a design model, see [Høydalsvik and Sindre, 1993] for a discussion on the distinction between analysis and design.

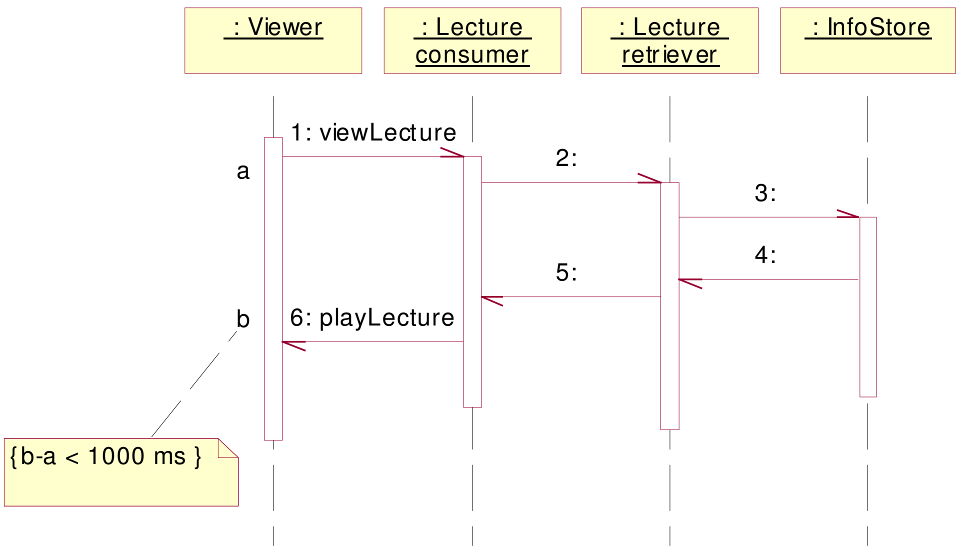


Figure 43: View lecture high-level sequence diagram

By marking two points in time on the sequence diagram, we identify the two events that can be at most 1 second apart. This shows in the model what the "usable" QoS statement in the "StudentQuality" QoS profile specifies. In addition, this model shows the high-level components involved in providing this service and hence, the components that contribute to the response time and that may be subject to QoS management. Message invocations 2 to 5 are anonymous; we only show that the interactions occur when providing the use case.

A 1 second response time may be tolerated for initiating a session to view a lecture, but for instance the user to possibly wait 1 second after requesting a stop-command until the video actually stops in the window is not acceptable. We therefore need to modify the "usable" QoS statement to distinguish between different operations of Lecture. In order to do this, we refine the Viewer to contain a Displayer that shows data units provided to it. The Viewer would in addition typically contain user interface logic, for instance for window placement, window content management etc.; we combine this into a GUI component.

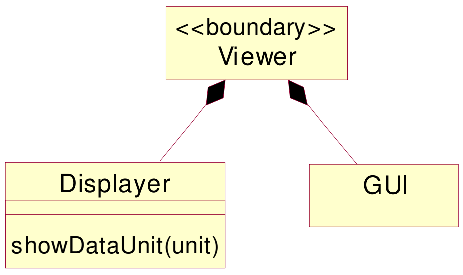


Figure 44: Viewer decomposition

A sequence diagram between a Displayer and a Lecture consumer illustrates how different Lecture operations handled by the Lecture consumer may have different constraints:

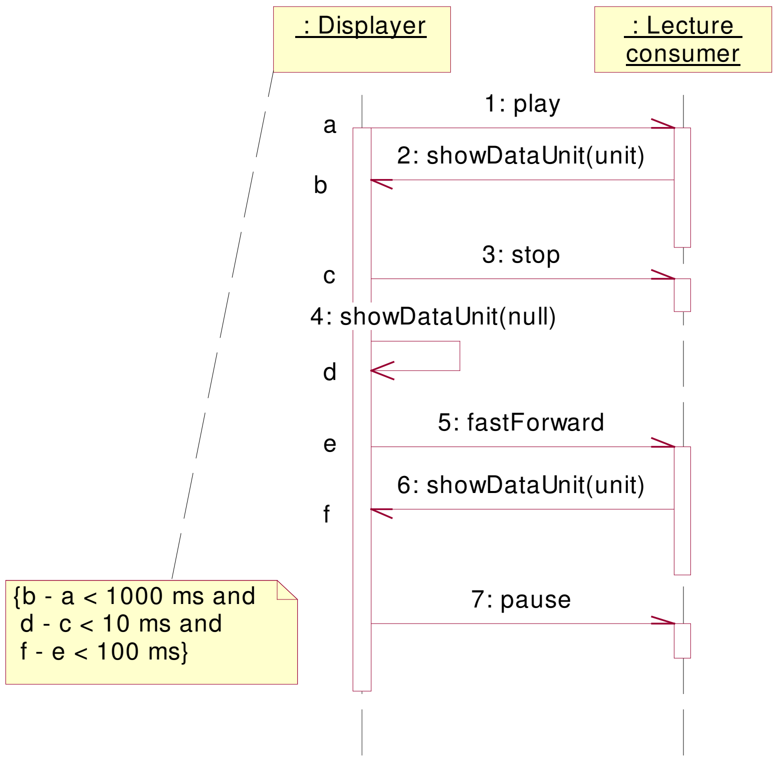


Figure 45: Displayer sequence diagram

A new "usable" QoS statement, capturing the constraints in Figure 45, illustrates how QoS constraints can be specified for individual operations:

```
quality usable(d : Displayer) {
    response_time(d.Lecture_Consumer.play, d.showDataUnit)
      < 1000 and
    response_time(d.Lecture_Consumer.stop, d.showDataUnit)
      < 10 and
    response_time(d.Lecture_Consumer.fastForward,
                    d.showDataUnit)  < 100;
}
```

Note that the pause-operation does not require any constraint since it simply tells the Lecture consumer to stop sending data units for the Displayer to show; the Displayer would then continue to display the latest data unit.

We have now refined the "usable" QoS statement to consider different response times for different operations, and have identified the high-level system components that are involved in providing the services for which usable response time is required. In addition, the "good_media" QoS statement specifies that the media elements should be smooth and have good quality. This is addressed in detailed design next.

5.5.6 LoD Detailed Design

The objective of detailed design is to refine the design model into sufficient level of detail so that it can be a useful basis for implementation.

5.5.6.1 Lecture Consumer

In addition to response time, media quality and smoothness must also be addressed. Media quality depends on the kind of media, typically QoS characteristics such as resolution and frequency are used in a refinement of media quality. Smoothness can be refined to depend on skew and jitter:

```
quality_characteristic smoothness {
   domain : increasing enum {poor, good};
   skew;
   jitter;
}
```

The Lecture consumer receives media flows that contain data units it delivers at suitable times to the Displayer. A detailed design model for the reception of flows is shown in Figure 46. A Lecture consumer contains buffers in which it receives data units. The data units are displayed through the Displayer interfaces.

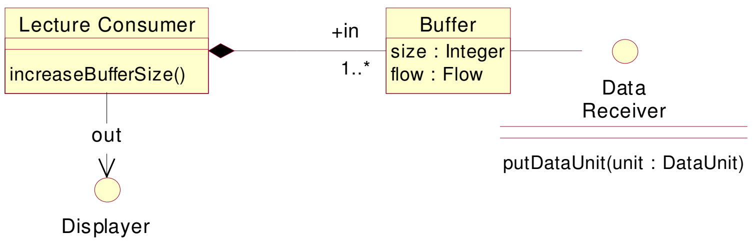


Figure 46: Reception of flows

The LoD system is not designed for interactive lecturing that requires real-time synchronisation between different locations. Hence, buffering is a viable solution to deal with jitter. The Lecture consumer should present its flows in a smooth manner (i.e., with little skew and jitter) and with good media quality (e.g., resolution, frequency, etc.). Since buffering is a viable solution, the Lecture consumer may increase the quality of the flows with respect to skew and jitter. The following is a profile for the Lecture consumer:

```
profile Acceptable for Lecture_consumer {
   profile Good {
     uses good_media(in.flow) and smoothish(in.flow);
     provides good_media(out) and smooth(out);
   }
   profile Ok {
     uses good_media(in.flow);
     provides good_media(out);
   }
   transition Good->Ok: increaseBufferSize();
}
```

Two profiles are specified with transitional behaviour between the Good and the Ok profiles. The adaptation scheme is to increase the buffer size if the smooth QoS offer is no longer provided in hope of the Good profile to become valid again.

Based on the UML profile defined in section 5.4, a UML model of the Acceptable QoS profile is shown in Figure 47.

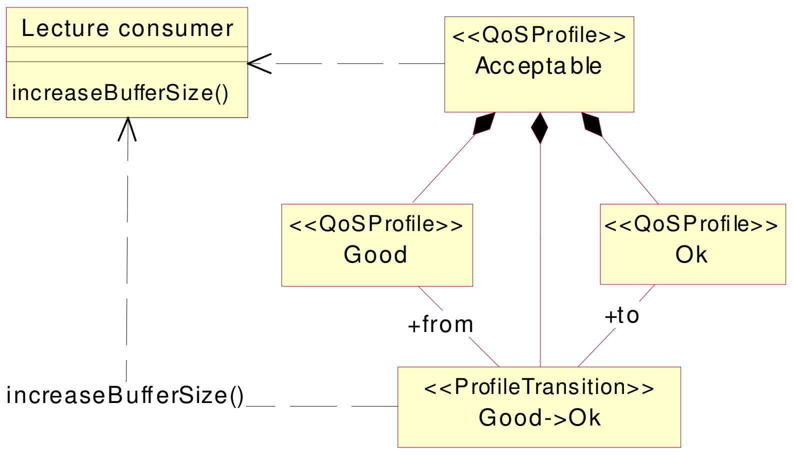


Figure 47: Acceptable QoS for Lecture consumer

The QoS offers of the two QoS profiles Good and Ok are shown in Figure 48.

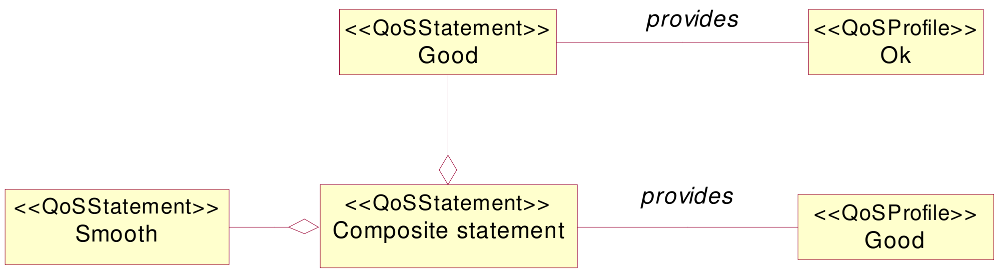


Figure 48: QoS offers of Ok and Good QoS profiles

In order to specify the QoS statements, we first need a model of the multimedia stream that the Lecture consumer processes. A stream consists of a number of Flows that in turn contains a number of data units. A Flow requires a DataReceiver interface to put the data units as they arrive. The arrivals of data units in the Flow objects are not shown in this model.

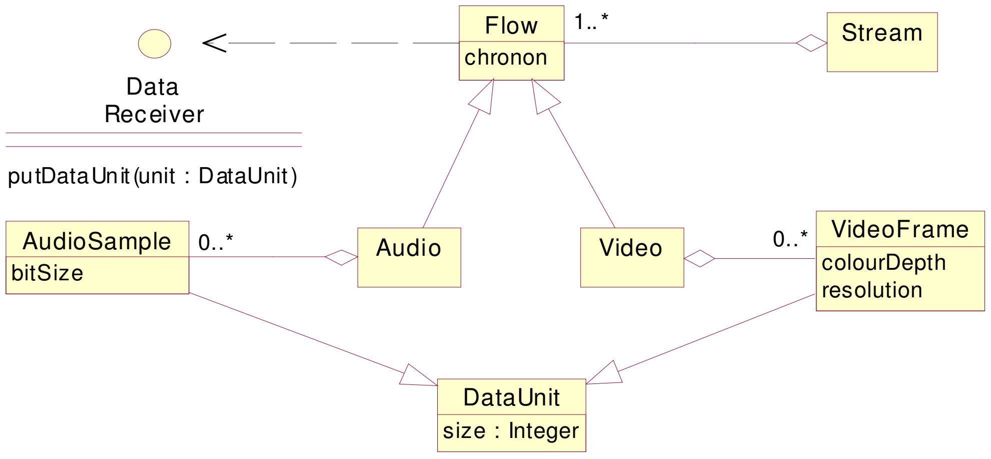


Figure 49: A model of multimedia streams

Based on this, the smoothness constraints are specified as:

```
quality smoothish(stream : Collection(Flow)) {
   jitter(stream).abs < 100 and
   skew(stream).abs < 250;
}
```

```
quality smooth(stream : Collection(Flow)) {
   jitter(stream).abs < 10 and
   skew(stream) < 120 and
   skew(stream) > -80;
}
```

The asymmetry of skew reflected in the `smooth` QoS statement is reported in [Engel and Steinmetz, 1993], where human perception of media synchronisation is studied. Note that this definition of `smooth` is a refinement of the QoS statement `smooth` in subsubsection 5.5.3.3 (where it was only specified that `smoothness = good`).

Jitter characterises intra-flow time variance of flow elements, and its QoS characteristic is defined as:

```
quality_characteristic jitter(flow : Flow) {
   domain : numeric integer milliseconds;
   values: if flow.SE->size > 1
           then flow.SE->last.time() - flow.SE->index(
                flow.SE->size - 1).time()) - flow.chronon
           else 0
           endif;
   composition: parallel-and if lhs.abs > rhs.abs
                             then lhs else rhs endif;
                parallel-or if lhs.abs < rhs.abs
                            then lhs else rhs endif;
                sequential lhs + rhs;
}
```

Note that the QoS characteristic jitter has a single Flow as parameter whereas it is used on collections of Flows. This is possible since jitter has specified composition semantics. Since both `in.Flow` and `out` used in the `Acceptable` QoS profile are aggregates, jitter for the composition is specified as maximum of the jitter for the parts (parallel-and). Alternatively, we can overload the QoS characteristic to have a collection of Flows as parameter and derive the maximum value for the collection in the values-clause:

```
quality_characteristic jitter(stream : Collection(Flow)) {
   domain : numeric integer milliseconds;
   values: stream->iterate(f : Flow; result : Integer = 0 |
                if jitter(f).abs > result.abs
                then jitter(f)
                else result
                endif);
}
```

Both ways of specifying the QoS characteristic give the same result.

The QoS characteristic skew characterises the duration between when a data unit is available and when it is supposed to be available, and is defined as:

```
quality_characteristic skew(flow : Flow) {
   domain : numeric integer milliseconds;
   values: flow.SE->last.time() -
          (flow.SE->last.play_time + flow.SE->first.time());
   composition: parallel-and if lhs.abs > rhs.abs
                            then lhs else rhs endif;
               parallel-or if lhs.abs < rhs.abs
                           then lhs else rhs endif;
               sequential lhs + rhs;
}
```

We use the `play_time` from our time model to denote when a data unit is supposed to be available. `play_time` is used to synchronise flows, i.e., all flow elements have a time when they are supposed to be available, and skew is therefore a measure for how well synchronised the flows are.

The profile for the Lecture consumer also contains the QoS statement `good_media` that has a Stream as parameter. That a Stream has good quality means that each Flow has good quality:

```
quality good_media(stream : Stream) {
   stream.Flow->forAll(f : Flow | good_media(f));
}
```

The `good_media` QoS statement is used for all kinds of flows in the Stream, but we need to distinguish between each flow type in order to define what characterises its goodness. We therefore overload the QoS statement to be defined for both audio and video:

```
quality good_media(f : Video) {
   frames_per_second(f) >= 15 and colour_depth(f) >= 16
   and resolution(f) >= 320x240;
}

quality good_media(f : Audio) {
   frequency(f) >= 16000 and bit_size(f) >= 16;
}
```

The QoS characteristic `frames_per_second` is defined as:

```
quality_characteristic frames_per_second (flow : Video)
          : rate(flow.SE) {
  domain: increasing numeric frames/second;
}
```

This definition is derived from a QoS characteristic `rate` defined as:

```
quality_characteristic rate(events : EventSequence) {
   domain : increasing numeric;
   values: events->eventsInRange(1000);
}
```

The last three QoS characteristics used in the `good_media` QoS statement are defined as:

```
quality_characteristic frequency(flow : Audio) :
          rate(flow.SE){
   domain : increasing numeric samples/second;
}

quality_characteristic bit_size(flow : Audio) {
  domain: increasing numeric bits;
  values: flow.DataReceiver.putDataUnit.SE->last.unit.bitSize;
}
```

```
quality_characteristic resolution(flow : Video) {
  domain : increasing enum{160x120, 176x144, 320x240, 640x480,
                     800x600, 1024x768, 1152x864, 1280x1024};
  values: flow.DataReceiver.putDataUnit.SE->
                                    last.unit.resolution;
}
```

5.5.6.2 Binding

In order to transport the multimedia information across the network in a timely manner, we need communication software with requirements on reliability, delay and throughput. From the business distribution model, we have a constraint that the available bandwidth will at least be, but cannot be assumed to exceed, the throughput capacity of ISDN, i.e., 128 kbit/s. In the structured system use case model in Figure 40, this constraint is modelled to pertain to the communication between the View lecture use case and its included use cases Find lecture and Retrieve lecture. In Figure 50, we show the model of a binding that realises the association between the general roles of source and sink, i.e., a producer and a consumer of a stream. The Binding realises the association between one source and one sink, so a source that has more than one sink will have more than one binding (multicast is treated as one logical sink). The model also shows that Lecture consumer plays the role of one or more sinks, i.e., it can receive several streams. Note that the binding both implements and uses the DataReceiver interface, this is because the binding must receive data units from the Source and put them at the Sink.

The binding is an example of an explicit binding aligned with RM-ODP and used in research on communication support for multimedia [Fitzpatrick et al., 1998, Plagemann et al., 1999, Plagemann et al., 2000].

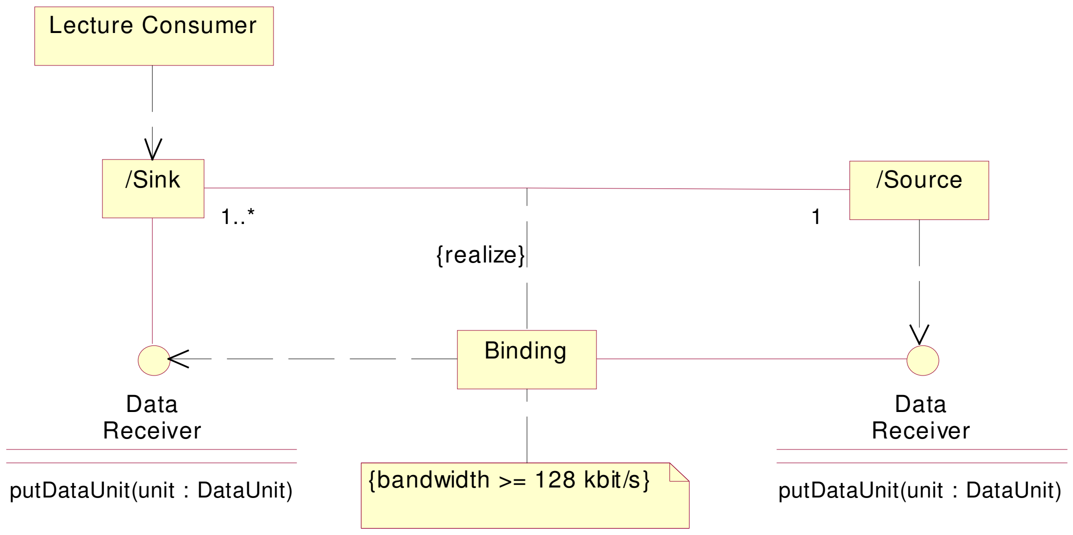

Lecture Consumer
/Sink
1..*
{realize}
/Source
1
Binding
Data Receiver
putDataUnit(unit : DataUnit)
{bandwidth >= 128 kbit/s}
Data Receiver
putDataUnit(unit : DataUnit)

Figure 50: Binding

The model shows that each Sink has a Binding that transfers the stream to it by invoking the putDataUnit operation. The following profile specifies that the binding should transfer the information fast, with low loss and have sufficient throughput:

```
profile Good for Binding {
  profile Starting {
     provides startUpConditions(putDataUnit);
  }
  profile Operational {
     provides lowLoss(putDataUnit, DataReceiver.putDataUnit)
       and fast(putDataUnit, DataReceiver.putDataUnit)
       and ISDNThroughput(putDataUnit,
      DataReceiver.putDataUnit,
                         putDataUnit.unit.size);
  }
}
```

Note that we use the operation putDataUnit both for the DataReceiver interface of the Binding and the DataReceiver interface of the Sink (referred to as putDataUnit and DataReceiver.putDataUnit, respectively). The DataReceiver interface is a generic interface for sending data units that does not distinguish between the different flows. This distinction can be made based on the context information of the data units, but it is not shown here.

During an initialisation phase, the Binding provides no services. Using the Clock from the time model in subsection 4.5.2 and constraining the initialisation duration to be maximum 1 second, this can be specified as:

```
quality startUpConditions(in : Operation) {
  Clock.now - in.SR->first.time() <= 1000;
}
```

Given that LoD is a multimedia application, some communication loss can be tolerated. We specify that the binding may loose up to 5% of its data units as carried by the operation parameters:

```
quality lowLoss(in : Operation, out : Operation) {
   lossRate(in, out) < 5;
}
```

This QoS statement specifies that only up to 5% of the data units provided to the binding in the in-operation are lost before being emitted through the out-operation. The specification of the values-clause in the lossRate QoS characteristic (see below) uses intersection between two event sequences produced at different locations (incoming and outgoing interfaces of the Binding) and hence, the intersection would always be empty. However, as discussed in subsubsection 4.6.5.2, we can use the identity of the data unit being transferred to decide whether two events are related and from this, calculate the intersection.

```
quality_characteristic lossRate(in : Operation,
                                out : Operation) {
   domain: decreasing numeric percentage;
   values: 100 - in.SR->subSequence(
                in.SR->size - 100, in.SR->size)->asBag()
                    ->intersection(out.SE->asBag())->size;
}
```

Note that lowLoss and any upper limit on the capacity restrict the size of the input to the Binding; since an output restriction is given by a throughput capacity and the Binding cannot discard more than 5% of its input, an input restriction follows.

The QoS statement that specifies the speed used in the QoS profile for Binding is:

```
quality fast(in : Operation, out : Operation) {
   delay(in, out) < 100;
}
```

The QoS characteristic used is specified as:

```
quality_characteristic delay(in : Operation, out : Operation)
{
   domain : decreasing numeric milliseconds;
   values: out.SE->last.time() - in.SR->last.time();
   composition: parallel-and lhs.max(rhs);
                parallel-or lhs.min(rhs);
                sequential lhs + rhs;
}
```

Note that the fundamental problem of synchronising distributed clocks poses restrictions on the calculation of the values of this QoS characteristic. However, measuring the time difference between an invocation and the acknowledgement that it has been received, divided by 2, can approximate the value. The emittance of the out-operation then acts as a trigger for the acknowledgement, and the calculation specified in the values-clause is thus a simplification.

From the business constraint on capacity in Figure 38, the throughput capacity of the binding will at least be the bandwidth available for ISDN, i.e., 128 kbit/s. The capacity is used to transfer data, so the size of the transferred data will at least be 128 kbit/s if sufficient input is provided:

```
quality ISDNThroughput(in : Operation,
                       out : Operation,
                       size : Integer) {
  rate(out.SE)*size >= 128000 or
  rate(out.SE)*size >= rate(in.SR)*size*lossRate(in, out);
}
```

This constraint may conflict with the QoS expectations from the Lecture consumer (good video and good audio); the Lecture consumer may want to transfer many flows of high quality while the Binding cannot be expected to transfer more than 128 kbit/s. We address this issue in the next subsubsection.

5.5.6.3 Communication

We have now specified the Binding and the Lecture consumer of the flows that the Binding transfers. The Binding has a lower limit of the throughput, but it has not yet been specified any upper limits on this. If no actions are taken, a good video flow requires 320x240 pixels * 15 fps * 16 bit/pixel = 18,432 kbit/s, which alone clearly exceeds the capacity of most communication channels. If the Lecture consumer in addition wants many video and audio flows transferred, design actions needs to be taken to resolve the conflict. The business constraint on the throughput capacity specifies that one cannot expect a bandwidth that exceeds 128 kbit/s. Hence, compression is useful. Figure 51 shows a design where the binding is refined to consist of encoders, a communication module and decoders. We assume that each flow within the stream may have a different encoder/decoder pair. The encoders encode the media information into a compressed format; the decoders reverse the process. If no compression is needed, the encoder/decoder pair is just forwarding the data.

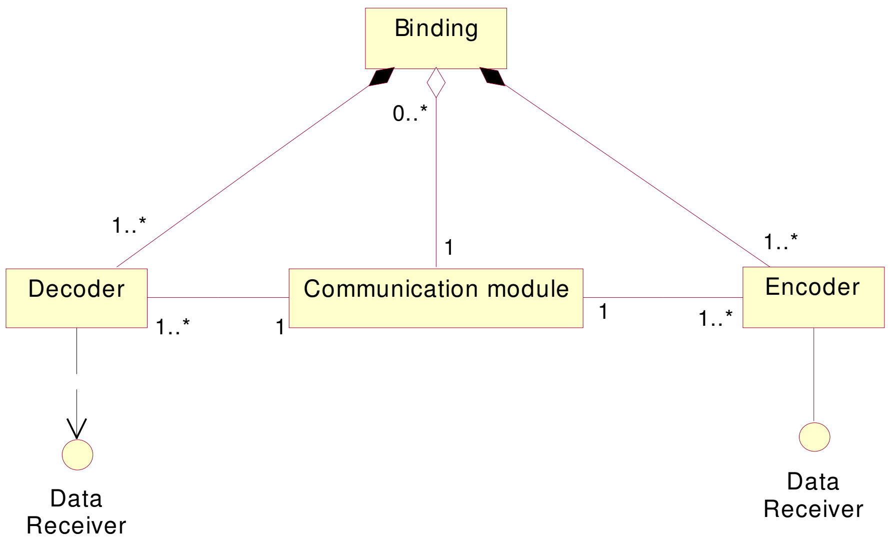


Figure 51: Binding refinement

The QoS profile for the Binding must collectively be satisfied by its constituents. The components of a Binding are used in sequence. Hence, the values of the QoS characteristics used in the QoS statements in the QoS profile for the Binding are composed using the sequential composition specification. For instance, the delay constraint that applies to the Binding must be honoured collectively by the three parts. The 200 milliseconds specified in the fast QoS statement used in the Operational QoS profile for Binding can for instance be divided into 10 milliseconds for encoding, 180 milliseconds for communication, and 10 milliseconds for decoding. We specify this as QoS profiles for each of the three components, that collectively refine the QoS profile for the Binding. We enclose the QoS profiles and the QoS statements into appropriate QoS categories to avoid name clashes:

```
quality_category Decoder {
   profile Good for Decoder {
     profile Starting {
          provides startUpConditions(putDataUnit);
     }
     profile Operational {
       provides lowLoss(putDataUnit, DataReceiver.putDataUnit)
         and fast(putDataUnit, DataReceiver.putDataUnit)
         and ISDNThroughput(putDataUnit,
          DataReceiver.putDataUnit, putDataUnit.unit.size);
     }
   }

   quality fast(in : Operation, out : Operation) {
     delay(in, out) < 10;
   }

   quality lowLoss(in : Operation, out : Operation) {
     lossRate(in, out) < 1;
   }
}

quality_category Communication {
   profile Good for Communication_module {
     profile Starting {
          provides startUpConditions(putDataUnit);
     }
```

```
      profile Operational {
        provides lowLoss(putDataUnit, DataReceiver.putDataUnit)
          and fast(putDataUnit, DataReceiver.putDataUnit)
          and ISDNThroughput(putDataUnit,
           DataReceiver.putDataUnit, putDataUnit.unit.size);
      }
    }
    quality fast(in : Operation, out : Operation) {
      delay(in, out) < 180;
    }
    quality lowLoss(in : Operation, out : Operation) {
      lossRate(in, out) < 3;
    }
}
```

The QoS specifications for Encoder are equal to those for Decoder and not repeated here. Also, the QoS statement ISDNThroughput for the three components is equal to its use for Binding and therefore not repeated here.

We assume here that even if the encoders and the decoders transform information, they intend to use the same number of input as output operations, the only difference being in the information content of the operations. In other words, information from several input operations is assumed to not be collected into a single output operation. Hence, since the definitions of the values-clauses depend on this, we can use the same QoS characteristics.

The loss rate has not specified compositional semantics, we need this in order to divide the loss rate of the Binding onto its individual parts. Loss rate QoS characteristic is extended to:

```
quality_characteristic lossRate(in : Operation,
                                out : Operation) {
   domain: decreasing numeric percentage;
   values: 100 - in.SR->subSequence(
                 in.SR->size - 100, in.SR->size)->asBag()
                     ->intersection(out.SE->asBag())->size;
   composition: parallel-and lhs.max(rhs);
                parallel-or lhs.min(rhs);
                sequential lhs + rhs - (lhs*rhs)/100;
}
```

The ISDNThroughput QoS statement provides a lower limit of the throughput of the Binding. This limit, 128 kbit/s, is in many cases sufficient to transfer an audio and a video flow with the specified quality. For instance, MPEG-4 [ISO/IEC JTC1/SC29/WG11, 2000] is a commonly used encoding format for media objects that compresses them suitable to be transferred by means of ISDN. E.g., using MPEG-4, it is possible to transfer 320x240 video with 15 fps and 16kHz mono audio on an ISDN connection.

However, even if one audio and one video flow do not saturate the communication module, it is a shared resource between all flows that may be saturated if more than one flow of each type is transferred. In subsubsection 6.4.1.1, a resource model is provided in which resources keep information about their capacity and the level of usage already reserved by others and therefore unavailable. In Figure 52, we show the Resource stereotype meta-class from subsubsection 6.4.1.1 that we use in the following.

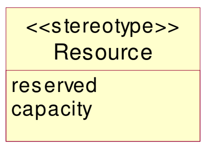


Figure 52: Resource stereotype

The shared Communication module can be viewed as a resource, and we model it as shown in Figure 53, using the Resource stereotype.

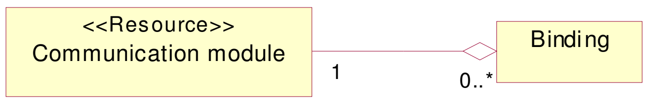


Figure 53: Communication module

In order to constrain the throughput of the Communication module with an upper limit corresponding to its capacity, we extend the ISDNThroughput QoS statement:

```
quality throughputCapacity (
        in : Operation,
        out : Operation,
        size : Integer) : ISDNThroughput(in, out, size) {
  rate(out.SE)*size <= capacity - reserved;
}
```

This QoS statement is a refinement of the QoS statement used in the QoS profile for the Communication module above, and we need to change the Operational QoS profile for the Communication module to include this new QoS statement:

```
   profile Operational {
       provides lowLoss(putDataUnit, DataReceiver.putDataUnit)
         and fast(putDataUnit, DataReceiver.putDataUnit)
         and throughputCapacity(putDataUnit,
          DataReceiver.putDataUnit, putDataUnit.unit.size);
   }
```

To initialise and update the values of capacity and reserved is a task for the underlying QoS framework, during QoS specification we only assume such values exist according to the resource model from subsubsection 6.4.1.1. The run-time dependant value of the upper limit on the throughput of the Communication module may propagate upwards in the design and violate QoS contracts that assume certain throughput such as "usable" for the LoD system. How this is handled is a task of the QoS framework, we have only specified dependencies in terms of QoS relations.

5.5.7 LoD Implementation

The detailed design of selected parts of the LoD system points to the need for an underlying QoS framework that provides QoS monitoring and QoS maintenance, for instance to manage resources such as the communication module. How specifications of QoS in CQML provide the information needed for QoS management is the topic of Chapter 6 presented next.

[Wang et al., 2001] reports a prototype implementation of a LoD system in the OMODIS project. This implementation focuses on an evaluation of the TOOMM multimedia model implemented in a multimedia database system, and does not integrate with a QoS framework or handle the QoS issues as discussed in the previous subsection.

5.6 Summary

In this chapter, we have shown that CQML can be used in different parts of software development. We have shown that it can be used in requirements engineering, to specify service level agreements, and in process assessment. We have also used CQML in a case study for development of a LoD system. In addition, we have specified a UML profile for QoS.

Part III. Computational Support

Chapter 6 CQML and a QoS Framework Architecture

The purpose of this chapter is to show that CQML supports specification of information needed for QoS management. In order to show this, we define a representative QoS framework architecture and point out how information originated from CQML specifications can be used for QoS management in this architecture.

6.1 Background

As mentioned in section 2.5, QoS management is a general term for activities to support configuring, monitoring, controlling and maintaining QoS. Such activities may be performed by many parts of a QoS-aware system in different architectural layers. Many isolated areas of QoS management have been addressed over the years, such as resource allocation in communication networks and in real-time operating systems, but with the recent attention on end-to-end support for QoS management, QoS frameworks that include QoS management activities in all parts of a system on all architectural layers have been proposed. In [Aurrecoechea et al., 1997], a survey of QoS architectures is presented. The surveyed architectures focus on communication and network issues, ignoring for instance QoS management in database systems. Our intention is to position concepts from CQML in a representative QoS framework that addresses salient aspects of the current QoS frameworks.

The architecture for a QoS framework presented herein is a straw man architecture designed as part of developing computational support for CQML. Other parts of the OMODIS project address this topic and the architecture presented here has been used as a starting point for the OMODIS QoS framework. A QoS framework should have scalability as a major design goal, in particular, it should be useful in a large-scale distributed system, within a complex multimedia database management system, and even in system elements such as storage managers. In addition to providing QoS management, the architecture presented herein has conformance with CQML as its primary concern in order to provide a validation of the concepts in the language. If we assume that CQML can be used to specify QoS on any level of abstraction for any type of component, the focus on conformance with CQML aligns with the scalability objective of the OMODIS QoS framework. Since the architecture targets to address the most important issues of current QoS frameworks, the use of CQML in the architecture shows, if the target is met, that CQML can support any of the current QoS frameworks.

6.2 Terminology

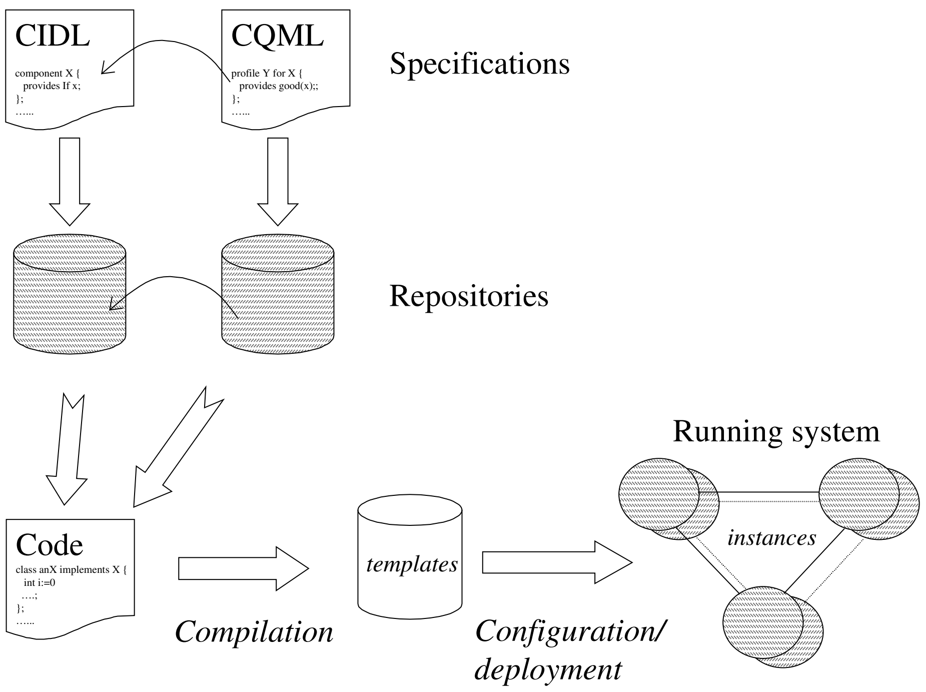
In order to be precise when presenting the QoS framework, we first define some terminology. The terms defined are not specific to QoS, they denote system concepts in general.

A software *system* consists of a set of *components* and some *infrastructure* on which the components exist. A set of components may comprise an *application*. A *component instance* is a physical unit of implementation and as such, it occupies some space in computer memory at run-time. A *component template* is a specification of the properties of a group of components so that individual component instances can be made. The components instantiated from the same component template have the same properties, but possibly with different values. We use the term component to mean both component template and component instance when it is clear from the context which we mean. An *interface* is a named set of operations that characterise the functional behaviour of an element such as a component. A component has well-defined interfaces. That is, it *offers* a set of services as defined by its interfaces. A component may *require* a set of services to be able to provide its services. Interfaces may characterise both offered and required behaviour, and are termed *receiving* and *calling* interfaces, respectively. In many cases, these are also referred to as *incoming* and *outgoing* interfaces, respectively. A component instance (client) may collaborate with another component instance (server) by invoking operations the server offers in a receiving interface. The client sends its invocations through a calling interface. For such a collaboration to take place, a connector between the client and the server is needed. We do not distinguish between components and connectors, instead we use binding objects that are instances of a special type of component, to model communication between application components. A *contract* denotes a constrained collaboration between two component instances, i.e., a sequence of operation invocations between the two component instances subject to collaboration-specific constraints such as postconditions, QoS constraints, etc.

6.3 Application Development

All applications must be specified before they can be executed. Such specifications can be divided between specification of functional and non-functional properties. A QoS-aware application would need both types of specifications, but whether these are combined into an all-encompassing specification or whether the two aspects are specified separately is decided by the capabilities of the specification languages and the chosen architectural style. As already mentioned, we believe an orthogonal approach where the two aspects are separated is preferable since it facilitates minimal intrusion of existing systems if they are to become QoS aware. Separation of concerns is also a widely accepted principle in software engineering.

Specification of the functional properties is not addressed here, but we assume such a specification exists and it contains specifications of components that collaborate to provide the application properties. We also assume specifications of non-functional properties are using CQML as their specification language. In Figure 54, we illustrate an application development scenario where functional properties are specified in CIDL and non-functional properties are specified in CQML. The CQML-specification refers to the component specifications in CIDL. Both specifications are stored in repositories. The specified system is implemented by writing code in an appropriate language based on the specifications of system properties in CIDL and CQML. The implementation code is compiled into a repository of component templates. These templates can be configured and deployed into a running system. The component instances in the running system have both functional and non-functional aspects satisfying the two specifications.



```
component X {
  provides If x;
};
……
```

```
profile Y for X {
  provides good(x);;
};
……
```

```
class anX implements X {
  int i:=0
  ….;
};
……
```

Figure 54: System development

The specification repositories may be used during compilation to check that the implemented components have the appropriate properties; they also contain information about the properties of the running system that can be used during run-time. For instance, components may use the repositories to get information about themselves (akin to reflection) in order to adapt to the current conditions.

6.3.1 Infrastructure Support

Activities in the infrastructure to support applications can be categorised into three phases indicating when they may be performed: prediction phase, establishment phase, and operational phase. In the prediction phase, aspects of system behaviour are estimated enabling appropriate mechanisms to be initiated later. For instance, the predication phase could include inquiries of resource loads combined with inquires of application resource requirements. Based on these, estimates can be made of how well the system can behave under certain conditions. These predications can be used during later phases as the basis for configuration and adaptation decisions. Activities in the establishment phase include setting up the conditions so that the system can behave properly (including, for instance, resource reservation) and initialisation of mechanisms that will be needed during the operational phase. The operational phase includes monitoring and maintenance (e.g., adaptation) of system behaviour.

The draft ISO specification on QoS in ODP extends the three-phase categorisation into five stages of an activity:

- a priori – appropriate functionality built into the system at design-time, e.g., by dedicating resources, systems sizing, etc;
- before initiation (corresponds to the predication phase) – requirements are conveyed to application components before initiation so that appropriate resources can be reserved at the initiation;
- at the initiation (corresponds to the establishment phase) – requirements are negotiated between provider and consumer, and required resources are reserved;

- during the activity (corresponds to the operational phase) – the environment may change so that re-negotiation of the initial contracts may be required or local adaptation can be performed to continue performance under the current contract;
- historically – statistical analysis of the activity, such as trend analysis, contract evaluation, future predictions, etc.

Even though this categorisation is defined to classify activities in a QoS framework, it applies equally well to infrastructure support for the functional parts of an application.

During these stages, the repositories play an integral role in facilitating the infrastructure to perform supporting activities. Relevant information available a priori (e.g., based on system specifications) is stored in the repositories. The specifications in CIDL and CQML in Figure 54 represent such information. Before initiation (in the prediction phase), the repositories are used by the infrastructure to identify possible configurations and their properties so that when initiating the activity (in the establishment phase), an appropriate configuration can be chosen (e.g., from a prioritised list of possible configurations). During the activity (in the operational phase), the infrastructure may need to consult the repositories if adaptation is required, otherwise the application will operate without concerns of how it is structured and which alternative configurations exist. If historic analysis is supported, the infrastructure needs to report appropriate system conditions during operation. The level of involvement in the different stages depends on the capabilities of the infrastructure. The minimal solution is to hardwire the configuration during design and only keep this in the repository. The repository is not involved in the operation of such a system, therefore the capabilities of such a system would be limited.

6.3.2 Relevant Approaches

When configuring an application, information about both functional and non-functional aspects are needed. Approaches already exist where information is stored to support configuration and adaptation based on functional aspects only. Here we discuss two such relevant approaches: trading and architecture description languages (ADLs). They do not focus in particular on QoS, but are used in general system development and deployment.

6.3.2.1 Trading

For distributed systems, component interfaces are often specified in an interface definition language (IDL) like OMG IDL [OMG, 1996]. Using IDL, one can specify syntactically the functional interface of components. Following the terminology from RM-ODP, when using IDL we specify interface types that any number of objects may provide. Such interface types can be organised in a type hierarchy. Using such a hierarchy, a type management system can identify compatible types (by subtype or equivalence relations). Such a type management system is an important part of the trading function as specified within the context of RM-ODP [ISO/IEC JTC1/SC21, 1997] and later adopted in OMG [OMG, 1997]. Traders (implementing the trading function) are essentially match-makers between service offers and service requests. A service is specified by an interface and a number of instance-specific properties. Such service properties describe non-functional aspects (including QoS) of the service and are defined as a sequence of name-value pairs.

Standard trading approaches have received some criticism, e.g., as reported in [Vasudevan, 1998b, Vasudevan, 1998a, Popien and Meyer, 1994, Puder et al., 1995], where one of the drawbacks pointed out is that standard trading facilities are too static to handle configurable, active clients and servers (and configurable, active bindings between them) with time-variant behaviour. Adaptive applications exhibit such characteristics and therefore require enhanced trading functionality. A further drawback is that traders require service providers to specify what the services offer without having any possibility to make constraints about when the offer can be met. Such unconditional service offers are not adequate when a service implementation depends on other services to be available. An end-to-end QoS guarantee may involve both client and server (and any intermediate objects such as communication channels), hence, the model of pure service offers is not adequate. We therefore need an extension of traditional trading functionality in our QoS framework. Another drawback with the traditional trading approach is that the non-functional properties are only specified as name-value pairs (tagged values), there is no underlying semantics of what each value actually represents. To rectify this, we use information from the models of the system and specify QoS properties with a semantic base using CQML.

6.3.2.2 Architecture Description Languages

In subsection 4.4.2, architecture description languages (ADLs) were introduced and their support for QoS specification were discussed. Even if they were evaluated to be, as they currently stand, not well suited for QoS specification, they are nevertheless useful for specification of component configurations.

ADLs are able to represent many of the aspects of system components a QoS framework needs to manage. The general system configuration is of great importance as components are the ones implementing the services, and their dependencies as defined in the configuration graphs constitute potential QoS dependencies. We want to enhance the representation of components and their configuration graphs to include specification of QoS by including QoS-related dependencies. Also, we want to extend the configuration graphs to include not only the actual configuration of a running system, but also include alternative configurations. This information can be utilised by adaptive systems when doing dynamic re-configuration.

6.3.3 Dependencies

As we describe next, a number of types of dependencies exist between components and interfaces and between interfaces in a system. These dependencies are useful for the infrastructure to be aware of in order to be able to configure and adapt applications.

- A component *offers* a set of services at a receiving interface. A component may implement a number of receiving interfaces, so a component may offer a set of services at a set of interfaces. The set of services offered by a component may change dynamically; offers may be revoked permanently if changes in the condition of operation invalidates some of the assumptions, or offers may be marked unavailable if temporary changes in the conditions of operation necessitate it.
- A component *requires* a set of services at a calling interface. This dependency means that a component may use services other components offer in order to function properly. A required service may be offered by many components at many receiving interfaces.
- In a running instance of an application, one of the required calling interfaces for a component instance is matched by an offered receiving interface of another component instance, i.e., the component instance *uses* another component instance. The common client/server model implies a uses-dependency between client and server.
- For a uses-dependency to exist, there must be a *partial_match* between the expectation of a component template and the offer of another component template. In other words, parts of what is required by one component must be provided by the other component if a uses-dependency is to occur. A partial_match-dependency between two component templates does not necessarily mean that instances of the two templates actually collaborate (i.e., have a uses-dependency), it only means that they can collaborate.
- *Complete_match* is a stronger version of partial_match in which all parts of the expectation of a component template is met by the offer of another component template.
- A component may *consist_of* a number of other components (parts). Often an aggregate component offers a set of services on behalf of its parts, i.e., it is a façade. A de-

composition of an aggregate into its constituent parts represents a shift in level of abstraction.

- A component instance *indirectly_affects* another component instance if it uses a limited resource at the same time as the other component instance is, or wants to be, using it. Note that it is only in the case of limited resources that this is relevant. If the common resource is unlimited, resource availability remains unchanged and the component instances are unaffected in this respect by each other's presence.
- For an indirectly_affects-dependency to exist, the component template *indirectly_influences* the other component template. A component template indirectly_influences another component template if there exist a configuration such that an instance of the first component template indirectly_affects an instance of the other. An indirectly_influences-dependency only indicates that an indirectly_affects-dependency may exist between instances of the templates. System configuration can assign different resource instances to the two component instances making them unaffected in this respect by each other.
- Conformance between interfaces is important to represent since substitutability is an important property of a reconfigurable system. An interface A is *conformant_to* another interface B if components implementing the interface A can be used in place of a component implementing the other interface B (adhering to the Liskov substitution principle [Liskov and Wing, 1994]).

In Figure 55, we have illustrated the offers- and requires-dependencies together with the partial_match- and complete_match-dependencies using UML. `Component1` realises two interfaces, `Interface1` and `Interface2`, i.e., it offers the services characterised therein. Similarly, `Component2` offers the services characterised in `Interface3` and `Interface4`. `Component3` uses `Interface5`, that is, it requires the services characterised therein. `Interface5` has a complete_match with `Interface3`, i.e., any component offering `Interface3` provides what `Component3` requires. `Interface2` has a partial_match with `Interface5` so that `Component1` can be used by `Component3`, but that will only partially meet its expectations. `Interface2` and `Interface3` may provide more services than `Interface5` specifies, but that is not of any concern. Note that we have used class symbols, not object symbols (with an underlined name). We use classes because we are not specifying individual instances, but rather component templates.

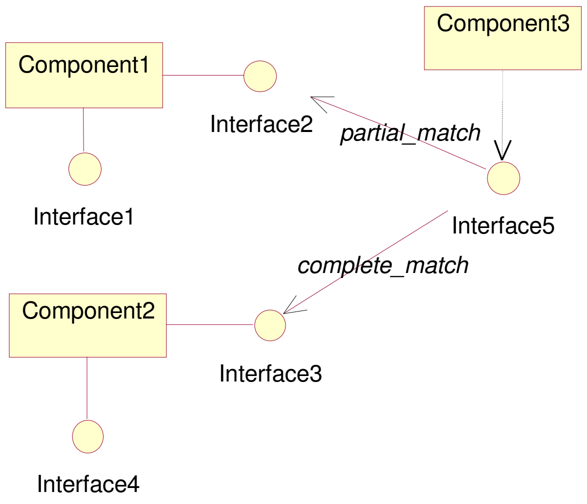


Figure 55: Dependencies

We have de-coupled interfaces implemented by components from interfaces required by components. This means that we can fully specify a component without referring to interfaces implemented by other components. The matching is a matter of configuration (dynamic or static). In Figure 55, `Component2` may be used to implement the required `Interface5` through its provided interface `Interface3`. `Component1` provides only parts of what is required, hence the partial_match-dependency between the provided `Interface2` and the required `Interface5`.

It is important to keep track of these dependencies in a running system if adaptation is an issue. The offers-, requires- complete_match-, and partial_match-dependencies determine the set of possible configurations, so an adaptive system may use this information to predict the results of changes in the environment and determine what configuration to chose if reconfiguration is required.

The consists_of-dependency is illustrated in Figure 56. The component `Aggregate` consists of `Part1` and `Part2`. Note that this figure represents two abstraction levels, one in which `Aggregate` is a component that implements `Interface1`, and one where `Aggregate` is decomposed into `Part1` and `Part2`. In the figure, `Part1` implements `Interface2`. `Interface2` is an internal interface in `Aggregate`, i.e., at the level of abstraction where `Aggregate` is used, `Interface2` is not visible. However, `Interface1` is visible here, and the figure illustrates that `Interface3` is conformant to `Interface1`. This can either mean that `Interface1` is an alias for `Interface3` so that `Aggregate` actually offers `Interface3` under another name, or `Aggregate` can offer the services characterised in `Interface1` by delegating behaviour to its constituent `Part2` which implements the conformant interface `Interface3`. Alternatively, `Aggregate` could have exported `Interface3` directly on behalf of `Part2`.

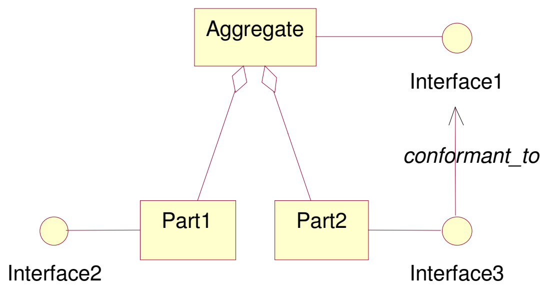


Figure 56: Aggregation

6.4 QoS Framework Architecture

6.4.1 QoS Management

A QoS framework provides assistance to an application in satisfying QoS requirements, that is, it performs QoS management. A prerequisite for QoS management is QoS specification, i.e., the capture of QoS requirements that the QoS management activities seek to assist in satisfying. In our context, QoS specification is done using CQML. Based on QoS specifications, QoS management can be performed during the different phases of application activities.

During the predication, establishment, and operational phases presented in subsection 6.3.1, different mechanisms are needed to perform the required QoS management activities. Activities that use these mechanisms may be initiated and performed by the application or the QoS framework. According to the ISO QoS framework, the following general mechanisms can be used during the prediction phase:

- “enquiries of historical information on QoS measures which reflect previous levels of QoS achieved”;
- “analysis of historical information on QoS measures which reflect previous levels of QoS achieved”;
- “prediction of QoS characteristics in the system (i.e. completion time)”;
- “calculation of potential perturbation if specific QoS requirements are requested and granted”;
- “evaluation of levels of QoS parameters to be requested in the establishment phase”;
- “checking that the requests will not conflict with admission control policies.”

The following general mechanisms, paraphrased from the ISO QoS framework, can be used during the establishment phase:

- negotiation mechanisms;
- resource allocation mechanisms;
- initialisations of operational phase mechanisms;
- synchronisation mechanisms, including mechanisms to distribute timing information, synchronise actions or events, and ensure other forms of coherence and consistency are implemented.

During the operational phase, the following mechanisms are identified in the ISO QoS framework:

- QoS monitoring;
- QoS maintenance, including resource allocation, admission control, and tuning;
- QoS enquiry;
- QoS alert.

The general QoS mechanisms listed above are from the ISO QoS framework that focuses on communication, but these general mechanisms are also applicable to non-network characteristics. Even if other mechanisms may be useful in certain contexts, we assume it is sufficient for a representative QoS framework to address the QoS mechanisms listed above. Hence, the target architecture presented herein uses this as a list of requirements.

6.4.1.1 Resources

A resource is a component that provides one or more services to its clients, and for which the underlying physical limitations are of importance. A resource does not need to be a physical device, but it has a physical underpinning that is essential for its ability to provide the services. In a response to a request for proposals for modelling time-, schedulability-, and performance-related aspects of real-time systems by the OMG [OMG, 2000b], a general resource model is presented that models resources, the services they provide, and the usage of these services. In Figure 57, the basic elements of this resource model are depicted. This model illustrates that a resource provides a set of services with some QoS offers, and usage of these services can have QoS expectations.

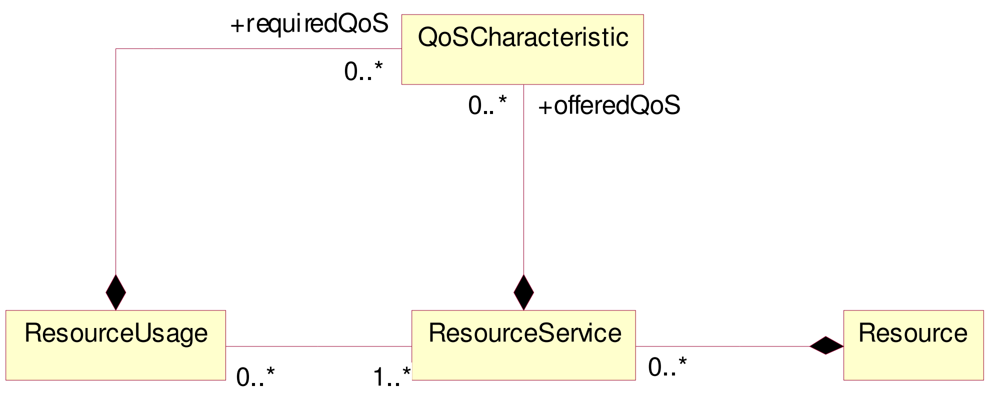


Figure 57: Basic resource model [OMG, 2000b]

We adopt the general resource model in [OMG, 2000b] and extend the notion of resource to have a scheduler and admission controller that admits requests into the scheduling queue. Clients request a resource by asking the admission control for access through the Reservation interface. If accepted, the request is put into the scheduling queue and processed as the scheduler sees fit. Figure 58 illustrates the extended model of a resource consisting of a Scheduler and an Admitter in addition to the resource model defined in [OMG, 2000b].

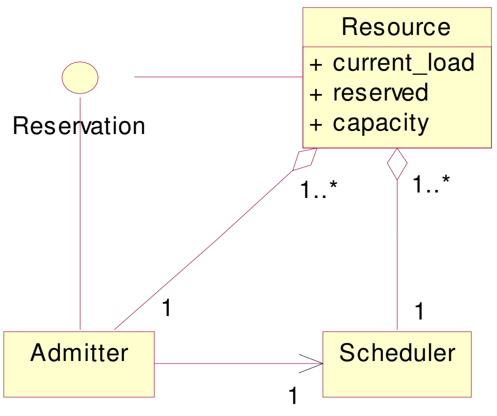


Figure 58: Resource extension

Both applications and the QoS framework may reserve resources as part of their activities. In particular when performing QoS management, resource reservation and admission control is important. However, we consider resource management not to be part of the responsibility of the QoS framework, the QoS framework only uses the resource management capabilities. Nevertheless, the resource management must provide appropriate support so that the QoS framework is able to perform QoS management. Each resource therefore needs to keep the following information (shown as attributes in Figure 58):

- current state of affairs with respect to use of the resource (load),
- level of usage already reserved and therefore unavailable, and
- theoretical maximum capacity.

The QoS capacity is a static value that is only changed when more resources are introduced into the system (in the case of an aggregate resource). The current state of affairs, on the other hand, is a very dynamic value that changes potentially whenever a component in the system changes state. Finally, the level of QoS already reserved is kept in order to be able to admit or reject requests for resource usage.

The QoS framework needs this information in order to be able to discover that adaptation is required and decide what adaptation strategies to use. Since the resource is a type of component, QoS profiles can be specified in CQML that define QoS offers of the resource services, i.e., the offeredQoS and the requiredQoS in the basic resource model in Figure 57 can be specified in CQML. This was for instance done for the Communication module in the LoD case study in subsubsection 5.5.6.3

6.4.2 Architecture Overview

In addition to performing QoS management activities as described in subsection 6.4.1, an important requirement of the QoS framework is that QoS-unaware systems should be unaffected by whether or not a QoS framework is part of the infrastructure. This ensures that legacy systems can be used unchanged. Furthermore, separation of concerns (orthogonality) suggests that we need to add components to the infrastructure rather than modify existing ones. We therefore choose to apply the façade pattern [Gamma et al., 1995] and expose all existing interfaces of the existing infrastructure through the façade. We also include interfaces pertinent to the QoS framework in the façade. Figure 59 illustrates the design where the infrastructure façade redirects requests to the appropriate part of the infrastructure. Note that the QoS framework uses the existing infrastructure, but not the other way around.

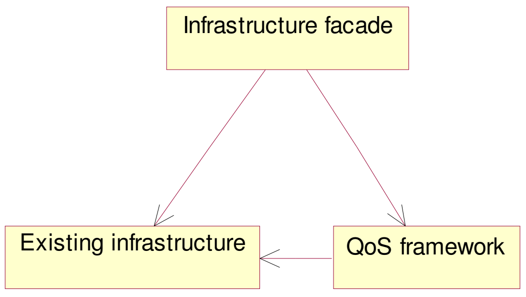


Figure 59: Infrastructure façade

To perform QoS management, we propose an architecture that consists of the following types of components:

- QoS Repository (QRep) – stores design specifications of QoS offers and QoS expectations of the individual components of the system, and how these relate to each other. It also stores historical information about component performance as reported by run-time measurements;
- QoS Monitor (QMon) – monitors system behaviour according to a set of constraints;
- QoS Manager (QMan) – co-ordinates QoS management activities, enforces system QoS policies, and initiates system behaviour such as user alerting;
- Adaptation Manager (AdaptMan) – initiates and performs adaptation of system components and their underlying infrastructure to achieve or sustain a specified QoS. As part of this, it estimates values of QoS characteristics based on historic values and hypothetical environmental conditions;
- QoS Negotiator (QNeg) – negotiates between service requestor and service provider to establish an agreement of service provision.

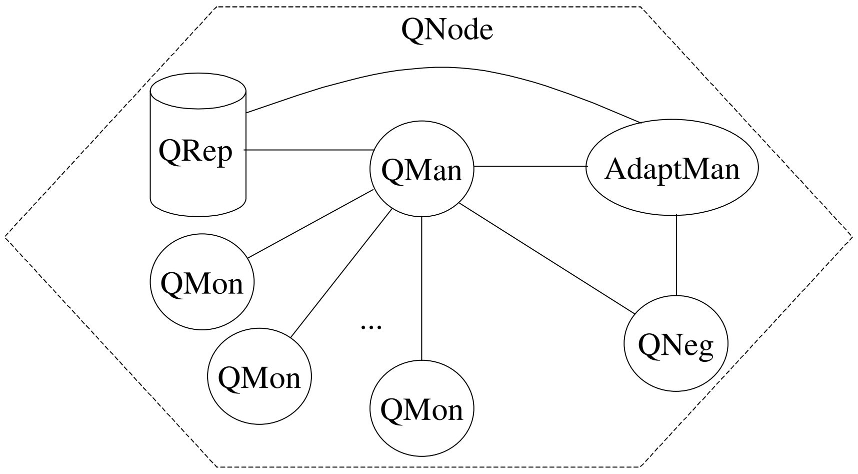


Figure 60: QoS node

The components are grouped into a QoS node (QNode) as depicted in Figure 60. During establishment, the QoS manager invokes the QoS negotiator to establish the initial contracts and it then initialises QoS monitors as required to monitor the new contracts. These QoS monitors report the current state of the QoS characteristics when needed. When adaptation is required, the QoS manager initiates the adaptation manager and delegates adaptation to it. The adaptation manager may use the QoS negotiator to establish new or revised QoS contracts. The QoS manager consults the QoS repository and reports run-time information. Even if QoS management is performed at all stages of system activity, these components are primarily used in the establishment and the operational phase.

A QoS node performs QoS management for a group of components. To ensure scalability, we divide the application components into domains following the approach reported in [Hafid, 1995, Hafid and Bochmann, 1998]. The notion of domain used by Hafid et al. corresponds to domain in RM-ODP, defined as a set of objects related by a characterising relationship to a controlling object. Hafid et al. present a hierarchical QoS framework architecture where each domain has a domain QoS manager that performs QoS management for all components in the domain, i.e., acts as the controlling object. This is applied recursively so that a domain may be an element of a higher level domain. Figure 61 illustrates this approach where a QoS node performs QoS management for a group of application components. An application component may be a composite component defining a domain where the constituting application components are managed by a separate QoS node.

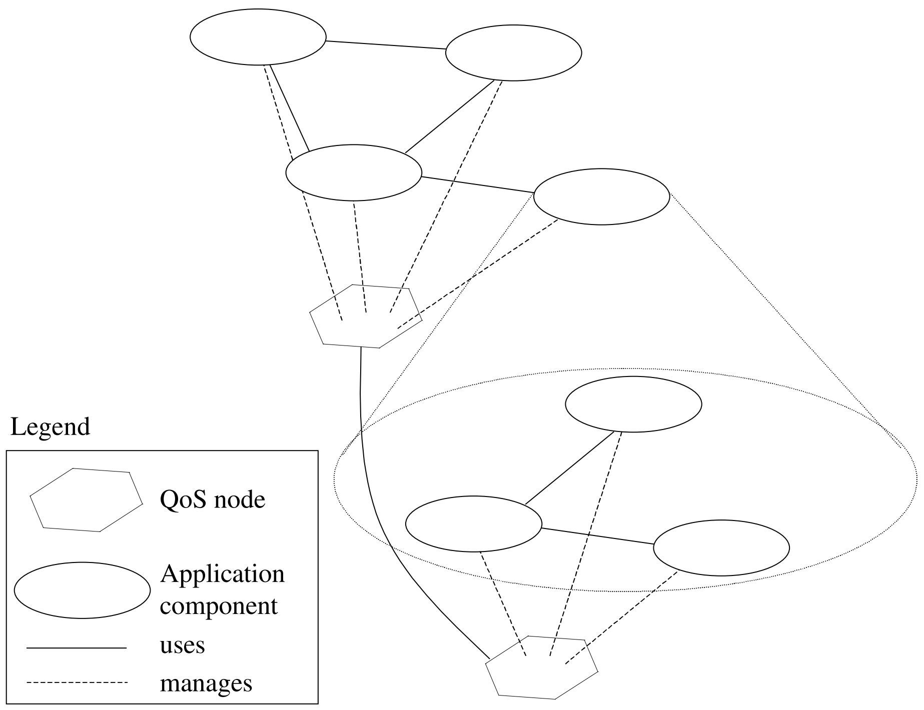


Figure 61: Hierarchical structure

In the following, we discuss the responsibility of each component.

6.5 QoS Repository

The QoS Repository (QRep) is a principal part of the QoS framework in the sense that it stores QoS information other parts of the QoS framework need in order to perform their designated activities throughout the activity phases. In addition to QoS specifications and historical run-time measurements for components, the repository contains derived information for individual components and for sets of components. Derived information includes validity of the specified QoS profiles, i.e., whether the QoS expectations can be provided (theoretically) by some components in the environment. It also includes conformance relationships between QoS profiles so that substitutions can be performed if necessary. Finally, the QoS repository stores some run-time information generated by other QoS framework components or the application components themselves. How each QoS framework component uses the information managed by QRep will be discussed in the sections describing the other QoS framework components. In this section we focus on what information is stored by the QRep and how that information is gathered or computed.

The dependencies identified in subsection 6.3.3 are relevant for QoS-unaware as well as QoS aware applications. For QoS-aware applications, however, additional dependencies exist that further constrain possible component configurations. These dependencies are:

- A component may offer a certain QoS for the services it offers. This is denoted a *QoS_offer*.
- A component may require certain QoS for the services it requires. This is denoted a *QoS_expectation*.
- For a uses-dependency to exist, there must be a *partial_QoS_match* between the QoS_expectation of a component template and the QoS_offer of another component template.

- *Complete_QoS_match* is a stronger version of partial_QoS_match in which all parts of the QoS_expectation of a component template is met by the QoS_offer of another component template.
- A QoS offer A is *QoS_conformant_to* another QoS offer B if components implementing the interface with QoS offer A can be used in place of a component implementing a conformant interface with QoS offer B.

To build an adaptive, QoS-aware application, the component specifications and their functional and non-functional dependencies need to be represented in an information store. For the functional dependencies identified in 6.3.3, traders and ADLs have already developed techniques and tools that are sufficient for their purposes. The information needed by ADL tools that is relevant in this context is a superset of that needed by the trading functionality, so the QRep is designed to use and augment the information available in ADL tools. Figure 55 illustrates some of the dependencies that ADL tools need to represent. Descriptions of components and their offered and required interfaces and the possibility to compute complete_match- and partial_match-dependencies between interfaces are required in an ADL tool. ADL tools and traders may provide implementation-dependant interfaces to retrieve this information, here we only rely on the information being available on a general basis. The QRep adds QoS-related information to this as described next.

6.5.1 Design-time Information

The QoS repository stores QoS specifications, i.e., QoS profiles and their constituent parts specified in CQML at design-time. The QoS profiles contain QoS_offer and QoS_expectation dependencies that the QRep stores. A QoS offer in a QoS profile is a nominal offer as specified by a truthful component, i.e., an instance of this component is able to provide the offer under certain feasible circumstances (i.e., in some possible environments). Note that a component may chose to offer less than its capabilities. For instance, a network able to transfer 100kbit/s may chose to only offer 10kbit/s. QoS specifications are stored for component templates, i.e., all component instances of the type share the same properties. Since the QoS specification are related to component templates, QRep needs to store a unidirectional link from the QoS profiles to their component specifications in the ADL tool in order to be able to relate the two types of specifications.

6.5.2 Derived Information

Based on design-time information, QRep is able to derive information needed for QoS management. The QRep uses information from the ADL tool in addition to the design-time information when deriving the required information. Whether an eager or a lazy approach is used for deriving this information is an implementation-issue and not discussed here. Three kinds of information are derived as discussed next: validity of QoS offers, QoS substitutability and indirectly_influences-dependencies.

In order for a QoS offer to be valid, it must be possible to configure the system in such a way so that its QoS expectations, if any, are met by other valid QoS offers. The QRep therefore needs to store or be able to compute partial_QoS_match- and complete_QoS_match-dependencies between QoS offers and QoS expectations. The possible configurations of components are a subset of the possible configurations based on functional properties as represented in ADL tools. If there exists a configuration of the system that meets the QoS expectations in a QoS profile, it is marked as valid. Correspondingly, it is marked as invalid if no such valid configuration exists.

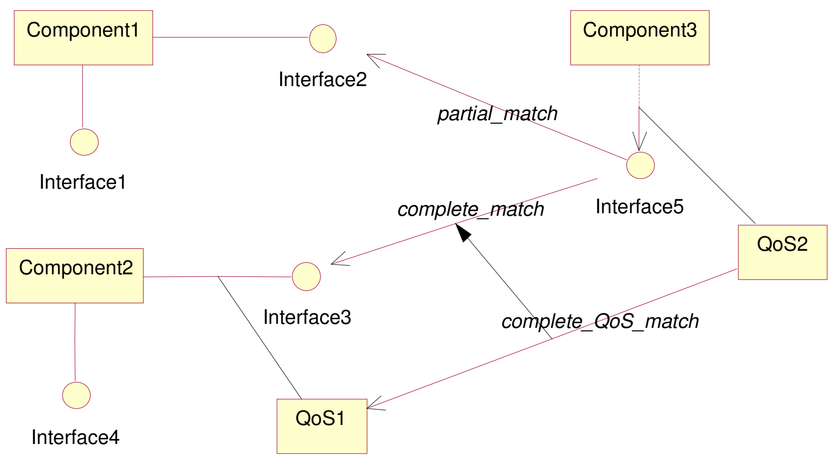


Figure 62: Dependencies with QoS constraints

Figure 62 shows a new version of Figure 55 where QoS has been added. In this figure, `Component2` provides `Interface3` with `QoS1` (QoS offer) while `Component3` requires `Interface5` with `QoS2` (QoS expectation).

A complete_QoS_match-dependency relates the QoS offer `QoS1` and the QoS expectation `QoS2`. Both complete_QoS_match- and partial_QoS_match-dependencies can be derived from CQML specifications as discussed in subsection 4.7.3. The complete_QoS_match- and partial_QoS_match-dependencies between QoS offers and QoS expectations are not useful without the function-based complete_match- and partial_match-dependencies stored by the ADL tool. This comes from the fact that QoS matching represents a further restriction of matching between components. Note that for `Component2`, we could have added another QoS offer `QoS3` that could act as an alternative QoS offer for the same interface `Interface3`. This would mean that `Component2` provides the same service with two different QoS offers. Note also that although we have shown the complete_match-, partial_match- and complete_QoS_match-dependencies to be between interfaces and between a QoS expectation and a QoS offer, the dependencies are actually between component templates providing or requiring the interfaces with the specific QoS properties.

In order to support QoS substitutability, the QRep needs to store or be able to compute QoS_conformant_to-dependencies between components. QoS substitutability means that a component can replace another without the environment noticing any degradation in QoS. In particular when performing adaptation, it is useful to know the substitutable components for the component that may need to be replaced since some of the substitutable components can have better QoS. As discussed in subsection 4.7.2, substitutability between components can be derived from CQML specifications using conformance between QoS profiles. However, the definition of substitutability requires a functional conformance relation between components. Hence, for QRep to be able to derive QoS_conformant_to dependencies and thereby QoS substitutability, it needs to access a corresponding conformant_to relation in the ADL tool.

The dependencies identified above as part of the content of QRep are based on a peer-interpretation of the collaborations between the components. The peer-interpretation means that the client and the server in a collaboration represent two distinct entities coexisting at run-time. Both the complete_match- and partial_match-dependencies depicted in Figure 62 are between two distinct component instances. Alternatively, dependencies exist that require a layer-interpretation of the model. The layer-interpretation means that the client and the server in a collaboration are not two different entities, but rather two different perspectives of the same entity. This is for instance the case when a component instance runs on a target platform and requires certain properties of this platform such as processor speed, memory size, etc. Indirectly_influences-dependencies between two components in a peer-interpreted collaboration may be discovered by investigating layer-interpreted collaborations where each of the two components acts as client. If the clients in two layered collaborations both require access to the same limited resource type, they indirectly influence each other. Indirectly_influences dependencies are important to represent in the QRep in order to support adaptation as will be discussed in section 6.8.

6.5.3 Run-time Information

There is a distinction between information produced at design-time and information produced at run-time. Design-time information as stored in QRep includes component template properties and defines potential system configurations. Run-time information, on the other hand, includes component instance configurations and measurements of system behaviour. Such measurements are in most cases not of interest outside the component instance in which the measurements are made and its QoS manager, and are therefore not stored in QRep. However, some run-time information may be reported to QRep to accumulate statistical information to facilitate historical analysis and prediction of QoS. What kind of information to report and how to report it depends on the capabilities of the monitors in the QoS framework. The frequency of updates depends on the frequency of property changes. In [Popien and Meyer, 1994], the authors discuss this for traders and propose a two-sort classification of properties where the properties classified as static are stored in the trader using a push-model while the trader uses a pull-model for properties classified as dynamic. The QRep can use a similar approach. For classification of properties, heuristics designed as part of the QoS framework or design-time involvement by a designer can be used, but this is not discussed further here.

Instead of explicitly storing run-time measurements, these measurements can be used to modify information in QRep. At run-time, measurements are made to determine the values of the QoS characteristics of component instances. Such measurements may reveal QoS offers that cannot be provided even if they are marked as valid in QRep. In such cases, the QoS offers should be marked temporarily invalid in QRep so care can be taken when establishing new contracts. This situation occurs for instance when a limited resource becomes unavailable so that some services no longer can be performed with the QoS offers stored in QRep. The QoS offers should be marked valid again in QRep when the prevailing system conditions change back so that its QoS expectations can be met. The validity of QoS offers for components depends on the validity of its QoS expectations. For resources, on the other hand, the validity of its QoS offers depends on the load. Since each resource keeps information about its load and the level currently reserved, the validity of the QoS offer of the resource may be derived from this and propagated to QRep to keep the validity of its QoS offers more current. In a 100kbit/s network, for example, the nominal offer of 10 kbit/s can no longer be valid if there are ten clients already having reserved a 10kbit/s portion. Note that best-effort requests can still be processed as the load may show that not all clients use what they have reserved, and others may use this on a best-effort basis.

QoS values for composites depend on the QoS values of the parts. Hence, the consist_of-dependencies in the ADL tools are also important in order to calculate QoS values of the composite. In CQML, compositional properties for different types of composition of the individual QoS characteristics can be specified, enabling QRep to calculate aggregate QoS when evaluating validity based on run-time information.

The indirectly_affects-dependency is especially important to support adaptation. Often there are limited resources available, and some co-operation is needed between components requiring the same limited resource. Scheduling and admission control are means of achieving shared use of resources. However, if two component instances using the same resource instance are part of the same system, a trade-off may exist between the usage. By granting permission to use the resource more heavily to one component instance than the other, overall system performance may increase if this component instance is more important for overall system performance. Such trade-offs can only be calculated if indirectly_affects-dependencies are kept track of. Predictions of system performance can be made for hypothetical system configurations using indirectly_influences-dependencies. Even if the indirectly_affects- and indirectly_influences-dependencies come from functional component specifications, we cannot assume that the ADL tool stores them as they are irrelevant from a functional point of view, and we must therefore include them in QRep. Indirectly_influences can be derived from the component specifications at design-time by recursively traversing the uses-clauses of the two component specifications that are being investigated and establishing a dependency if any of the services reached in the traversals originating from the two components are offered by the same component template. Similarly, the indirectly_affects dependencies can be derived in the establishment phase by investigating the identities of the component instances that are configured to provide the services identified in indirectly_influences-dependencies.

It is important to represent the consequences of restricting access to a common resource. This can help the QoS framework when deciding who is to get access to the resources. The consequences can partly be derived from the values-specification for the QoS characteristic since these specify how the properties of the component contribute to the value of the QoS characteristic. Properties of the underlying resource may influence the aggregated property of the components, giving the QoS framework an opportunity to derive the consequences of changing resource availability.

Figure 63 illustrates the indirectly_affects-dependency between two component instances both using the same resource instance. Calculating the trade-offs that may exist between these component instances can be very complex, but very important for adaptation. We assume that such trade-offs can be calculated by the adaptation manager, this is not part of QRep.

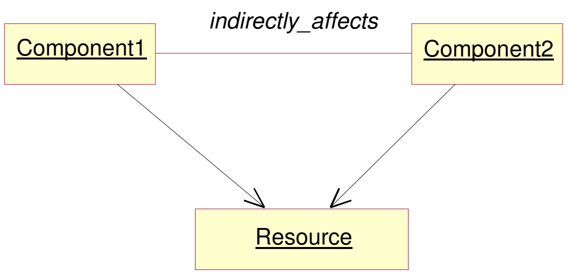


Figure 63: Indirectly_affects-dependency

6.5.4 QRep Hierarchy

The QoS repository is a logical unit, not a physical one. To ensure scalability, an important requirement of the repository is that it should not hold too much information in one place. The global set of QoS specifications and dependencies between components can be very large; we therefore need a partitioning of the QoS specifications and a delegation of these to individual and autonomous QReps. The interworking repositories that constitute the logical global QRep are called federated repositories. The individual repositories will have a limited view of the world comprised of the QoS information in its assigned domain. This is the same architecture chosen for the Trading Service in OMG and ODP [ISO/IEC JTC1/SC21, 1997, OMG, 1997].

The partitioning of QoS information into the individual repositories is guided by the general object-oriented principle of encapsulation. In principle, each component or resource may constitute a domain and have its own QRep where QoS dependencies between internal parts (if any) are represented, and the QoS offers and QoS expectations of the component or resource in question are accumulated. This can then be used at another level of abstraction as a means of deriving higher level QoS offers and QoS expectations in a higher level QRep. In other words, each QoS node uses its own single QoS repository.

6.5.5 Example of QRep Content

The QoS specifications and dependencies in QRep can be represented as a labelled directed graph where the nodes are QoS profiles and the edges represent dependencies. The direction is from the QoS profile that includes the QoS expectation to the QoS profile that includes the matching QoS offer.

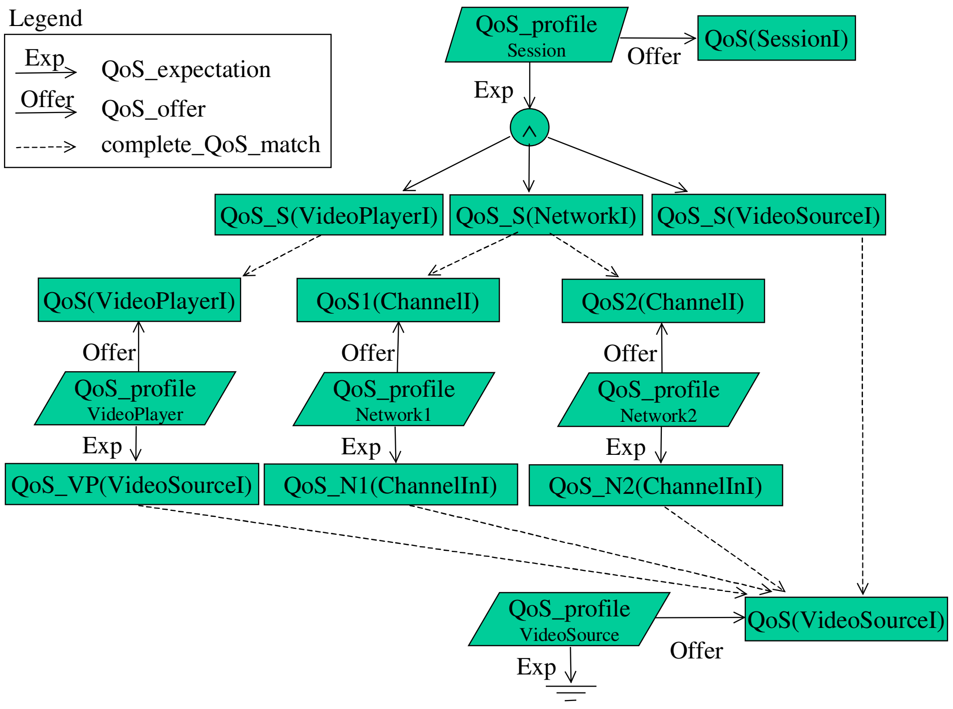


Figure 64: QoS repository content

In Figure 64, we show a directed graph corresponding to the information available from the QReps for an example video session consisting of a video source, a network and a video player. Two types of networks exist that both match the network type that the session requires. The QoS profiles are shown as parallelograms while the QoS expectations and offers are shown as rectangles connected with their QoS profile through the Exp and Offer relations, respectively. To illustrate that the sum of partial_matches is sufficient for the QoS expectation of the session, we have divided the QoS expectation of the QoS profile into three parts (using the AND-symbol '∧'), where each part has a matching QoS offer. Dashed arrows illustrate complete matches between required and provided QoS properties. The QoS profile of the VideoSource has a trivially valid expectation; we use a grounding symbol to illustrate this.

Each QRep has responsibility for the complete_match- and partial_match-dependencies that connect expectations of the component with offers from other components. In addition, the QReps need to know the indirectly_influences-dependencies between the components it has a complete_match- or partial_match-dependency to. For instance, to be able to tune the system in Figure 64 with regard to video quality, both VideoPlayer and Network1 may require the same resource (e.g., CPU). These two components may then influence each other. To maximise video quality, it may be smarter to let the VideoPlayer use the CPU than Network1, as the bandwidth may be the limiting factor of the performance of the Network1. Such indirectly_influences-dependencies specify that two components need a shared resource.

However, to be able to tune a system, we need to know the effects of changed resource availability of a component with respect to a specific QoS characteristic.

An active session must select one network, but as both Network1 and Network2 satisfy its expectations, other considerations can be made (e.g., cost). However, if the QoS offer from Network1 drops below what VideoPlayer requires, the complete_match-dependency no longer represents a valid configuration. VideoPlayer can nevertheless be operative, as Network2 still has a QoS offer that matches the requirements. However, if the QoS offer from VideoPlayer is reduced below what Session requires, the conjunction of expectations of the Session would no longer be valid, and Session would no longer be able to deliver its service with the offered QoS.

6.6 QoS Monitor

A QoS monitor watches system performance at run-time and reports the value measured or evaluates the value according to some constraint. The constraint is based on QoS offers in a QoS contract and is given to the QoS monitor as part of its initialisation. Hence, QoS monitors are used to check whether component instances deliver their promised QoS offers. A QoS monitor notifies interested parties whenever a violation of its constraint occurs. Typically, the initiator of the QoS monitor is interested in such notifications. The initiator may be another QoS monitor or a QoS manager. A QoS monitor may delegate responsibility to evaluate the constraint by instantiating new QoS monitors with parts of its constraint as an initialising parameter. However, in an instantiation-tree of QoS monitors, the root is always a QoS manager who can act upon the information.

A QoS monitor gets its run-time information as events generated by the component instances under observation. If, for instance, the QoS monitor for a binding object wants to check delay in the object's network connection, it gets an event from the network when a data unit enters the network at the source location and a new event when an acknowledgement of reception from the destination is received at the source location. From these events, the monitor can calculate the delay (½ * time-of-roundtrip) and compare the real delay against the promised delay in the QoS offer for the binding object. Statistical offers may involve accumulation of events to perform statistical analysis of event distribution. Note that even if a QoS monitor uses events resulting from system behaviour to evaluate its constraint, it does not have to be the timeliness aspects of the events that the QoS monitor is interested in. Hence, this does not preclude monitoring of non-time related QoS characteristics such as accuracy and security.

Instead of reporting constraint violations, a QoS monitor may report QoS values. This is useful for complex constraints that require evaluation of several QoS values. Each individual QoS value can then be reported to the initiator of the QoS monitor that combines these and evaluates the complex constraint. For instance, accumulated delay involving several components in sequence requires that each component that is part of the chain has a local QoS monitor that reports its delay value to a session-level QoS monitor who calculates total delay and evaluates the session-level constraint for delay.

6.7 QoS Manager

Each QNode has a QoS manager that is responsible for initiating and co-ordinating QoS management activities. A QoS manager retrieves the appropriate CQML profile that contains QoS offers and QoS expectations for its components from QRep, and initiates QoS monitors that watch the values of the relevant QoS characteristics.

To co-ordinate the components, the QoS manager initiates an appropriate QoS monitor (possibly hierarchically structured) for the QNode and communicates with the QoS managers of each component so that the QoS monitor for the QNode gets the QoS values or the violations from the QoS monitors of the components. The values of the QoS characteristics for the QNode are calculated in the QoS monitor based on the values for each component using the specification of the compositional semantics defined in the CQML specification of the QNode's component. For example, in the case of a session comprised of a client, a network channel and a server, the QoS monitor initiated by the QoS manager for the session would calculate the delay to be the sum of delays of each constituent part, while the overall throughput would be the minimum throughput of its constituent parts.

The QoS manager of a component is responsible for mapping QoS properties from one unit to another, if necessary. A simple example of such a mapping is when one part expresses its delay in milliseconds while another in seconds. A more complex mapping would be to map from bit per second to frames per second in a video flow. It delegates this mapping responsibility to the QoS monitor that reports the values by initialising it with appropriate knowledge on how to perform the mapping. This knowledge can be specified in the values-clause in CQML for the QoS characteristic. The QoS monitor, using this QoS characteristic definition, then reports its results in the proper metric. Alternatively, the QoS manager may use the unit specification for the QoS characteristic to determine the QoS mapping. However, CQML has only unit specification syntax, but does not specify the semantics, so how to interpret the unit specifications is not defined in CQML. Any interpretations therefore need to be uniquely defined for QoS managers to use this consistently.

Whenever the QoS manager becomes aware that a component is unable to provide its QoS offer, the QoS manager may update QRep to mark the QoS offer as temporarily invalid. For this, a QoS monitor informs the QoS manager that the constraint in the QoS offer is violated. As discussed in subsection 6.5.3, the opposite situation of valid QoS offers needs also to be reported to keep QRep current. The QoS manager also updates QRep with QoS values from its QoS monitors if analysis based on historical values is used as part of QoS management.

Based on reports from its QoS monitors, a QoS manager is responsible for initiating appropriate behaviour. Such behaviour may be identified in the specifications of CQML compound QoS profiles. Only behaviour that adheres to the system QoS policies can be initiated. Some such QoS policies may be specified in the invariants in CQML, and the QoS manager should continuously check the current state of affairs against these. However, CQML is not intended to be a policy description language and separate policy descriptions may need to be consulted. Nevertheless, different QoS categories can be specified in CQML that require different types of behaviour from the QoS manager. If a threshold QoS is specified, the QoS manager initiates user alerting. If compulsory QoS is specified, the QoS manager initiates service abortion. If possible adaptation is defined, the QoS manager initiates this. How the QoS manager initiates the behaviour and how it sees to it that it adheres to the system QoS policies is not discussed further here.

6.8 Adaptation Manager

The QoS manager initiates an adaptation manager to perform adaptation whenever the QoS manager sees fit. Typically this happens when a violation of the QoS offer of the component occurs or when the QoS manager has been requested to perform adaptation by other QoS managers. To select an adaptation strategy, the adaptation manager needs to retrieve from QRep the indirectly_affects-dependencies among the components it co-ordinates. It needs to know, for instance, that if it increases the frame-rate of a video flow, the overall capacity might make it necessary to reduce sample rate of the corresponding audio flow. The adaptation manager may also query QRep for historic QoS values to use for predicting component behaviour in order to choose the most appropriate alternative when performing adaptation.

If the QoS profile the QoS manager seeks to assist in satisfying is a compound QoS profile that contains a number of simple profiles with associated transitional behaviour, the adaptation strategy to use may already be determined by the transitional behaviour. The QoS manager invokes the operations specified as transitional behaviour whenever a transition occurs, these may trigger certain adaptation strategies in the adaptation manager. Such adaptation strategies may include increasing buffer size, modifying the time limit for data cache expiration, and even selecting a different algorithm if the component is modifiable. Where and how these strategies are stored and implemented is not pursued in this work, but CQML includes concepts to specify the conditions that define when adaptation is needed and what behaviour to initiate.

CQML also includes possibility to associate worth-values with the use of a service. For the adaptation manager, the overall task is to optimise the perceived value of the service. Hence, to choose what adaptation measures to take, the adaptation manager may use the worth-functions associated with the QoS profile in CQML.

6.9 QoS Negotiator

The QoS negotiator tries to match QoS expectations of one component with QoS offers of another component. The QoS manager initiates QoS negotiation between components that already are identified in the ADL tool as parts of a functionally valid configuration. For instance, in Figure 64, both networks provide the required functionality for the session and can be used in a collaboration. However, the QoS manager needs to consider their QoS properties and initiate negotiation between one of the network components and the session. The QoS manager decides which of the set of possible components to initiate negotiation with.

If a match can be made between the QoS expectations and the QoS offers, a contract is made between the requestor and the provider, and reported back to the QoS manager so that appropriate QoS monitors are instantiated in order to observe contract status. Note that a contract may specify further restrictions from the QoS offers; as long as the QoS expectations are met, a contract can be made.

QoS offers normally specify the lower bound of some QoS characteristics a component offers. If this is below the required level from the client, an immediate match is not possible. However, negotiation may lead to a contract where the service provider gives a better QoS than initially offered. For instance, assume a network with a capacity of 100kbit/s. From a policy decision, the network only offers 10kbit/s to any one client under normal circumstances. However, at low-peak hours, a request for 50kbit/s can be admitted, perhaps under the constraints that the bandwidth may be reduced if suddenly the unlikely event happens that other clients request bandwidth. However, if the client requests more than the capacity of 100kbit/s, the network cannot provide this and any negotiation would lead to slackened requirements from the client if a contract is to be established.

The negotiation process can become very complex, and significant research in distributed systems and AI has been applied to negotiation in multi-agent systems [Jennings, 2000, Zhang et al., 2000]. Also negotiation of QoS has received considerable attention [Koistinen and Seetharaman, 1998, Rothermel et al., 1997, Rajahalme et al., 1997, Hafid et al., 1996]. We do not focus of the negotiation process as such and will only use a simple matching to establish contracts. This does not preclude the use of more elaborate negotiation strategies at a later stage since the CQML specifications already include the aspects that are subject to negotiation.

6.10 Implementation

As part of work to implement a parser and a run-time representation of CQML, elements of a QoS framework have been implemented [Paulsen, 2001]. In addition to creating a compiler for CQML, the main focus of this work has been to design and implement the QoS repository since it is the component in which CQML specifications are stored and made available at run-time. The QoS repository has an interface specified in IDL in which operations for query, creation and manipulation of CQML specifications are included. The QoS repository also includes operations for conformance checks to enable application configuration and adaptation to be based on QoS information from CQML specifications. The initial implementation is not connected to any ADL tool, instead it uses a rudimentary ODL-based set of objects to represent application components (ODL - Object Definition Language [TINA-C, 1996]). The initial implementation does not include validity information nor derive the partial_QoS_match-, indirectly_influences and indirectly_affects-dependencies. However, to illustrate the use of the QoS repository, a rudimentary QoS negotiator was implemented that accesses the QoS repository and checks conformance between two CQML specifications in order to establish a QoS contract.

The QoS repository is implemented in C++ using micoORB [Römer et al., 2000] on a Linux platform.

Work is ongoing to implement other parts of the architecture presented herein [Lundby, 2001]. The main focus of this is the hierarchical structure of QoS nodes and in particular QoS monitors and their involvement in discovery of the need for adaptation. However, this work is still at an early stage and no results can be reported.

6.11 Summary and Conclusions

The QoS framework presented herein supports all activities identified for QoS frameworks in ISO QoS framework as presented in subsection 6.4.1. We have presented how CQML provides information required by the representative QoS framework to perform QoS management. Hence, CQML provides information required to perform QoS management on the assumption that the list of mechanisms from the ISO QoS framework is sufficient for a representative QoS framework.

The list of mechanisms required to perform QoS management identified in the ISO QoS framework is addressed by the architecture presented herein as follows.

- enquiries and analysis of historical measures – appropriate measured values are stored in QRep and accessed and analysed by QMan during initialisation and AdaptMan during adaptation;
- prediction of QoS – performed by QRep by deriving validity of QoS offers and by QMan by analysing statistical information;
- calculation of potential perturbation – performed by QMan based on indirectly_influences-dependencies in QRep ;
- evaluation of requested QoS – QMan uses validity-information from QRep to identify levels of QoS to request from individual components;
- checking against admission control policies – admission control is performed by the components according to the resource model outlined in subsubsection 6.4.1.1;
- negotiation – QMan initiates negotiation that is performed by QNeg. QNeg reports the established contracts back to QMan that supervises their fulfilment;
- resource reservation and allocation – initiated by QMan in the establishment phase and AdaptMan in the operational phase, and performed by the components according to the resource model outlined in subsubsection 6.4.1.1;
- initialisations – performed by QMan, for instance by initialising QoS monitors;
- synchronisation – performed by QMan;
- QoS monitoring – performed by the set of QoS monitors;
- QoS maintenance – QMan initiates adaptation performed by AdaptMan;

- QoS enquiry – performed by QMan to other QoS managers or QoS monitors if a pull-model of evaluation is used;
- QoS alert – performed by QMan based on notifications from QoS monitors.

The following summarise the use of CQML-specifications in the QoS framework:

- Domain specifications in CQML QoS characteristics define metrics the QoS monitors measure;
- Values-specifications in CQML QoS characteristics define what parts of the components the QoS monitors measure to compute the values;
- Composition-specifications in CQML QoS characteristics define how the monitored values are computed in a hierarchy of QoS nodes;
- Invariant-specifications in CQML QoS characteristics define QoS policies that the QoS manager enforces;
- Different types of QoS contracts can be agreed based on guaranteed, statistical or best-effort specifications in CQML. Different types of resource reservations can then be made by the QoS framework;
- CQML quality statements define the constraints the QoS monitors observe;
- CQML quality profiles specify QoS offers and QoS expectations that are needed to derive the conformant_to- and partial_match-dependencies;
- CQML facilitates adaptation by compound quality profiles that include several ordered profiles and the triggering of behaviour whenever a transition between these occurs;
- CQML facilitates adaptation by referencing worth-functions defined by clients that the adaptation manager can use to determine what QoS offer best suits the client.

It is important to note that CQML does not provide all information needed for QoS management. In particular, it is not a general policy description language or a negotiation protocol specification language. However, as discussed above, CQML provides support for specification of much of the information needed for QoS management.

The QoS framework herein has been presented and refined in [Ecklund et al., 2001a].

Part IV. Conclusions and Future Work

Chapter 7 Conclusions

This chapter contains a list of results and a summary of major contributions. It also contains an evaluation of the research method used in producing the results.

7.1 Claimed Results

In this section, claimed results are listed summarising what has been produced during the period of work. Claimed results include papers written and dissemination work carried out in the context of International Organization of Standardization (ISO) and Object Management Group (OMG). It also includes MSc-theses that have been supervised. The list includes the role the author had in making the contribution.

7.1.1 General Modelling

- "Role-based Modeling for the ODP viewpoints" – OOPSLA '96 workshop on Methodologies for Distributed Objects [Berre and Aagedal, 1996] – co-author
- "SIMOD - An ODP-extended Role-Modeling Methodology for Distributed Objects" – HICSS'30 [Berre et al., 1997] – co-author and presenter
- "Organization, Information System and Distribution Modeling: An Integrated Approach" – EDOC'97 [Silva et al., 1997] – co-author and presenter
- The OOram Meta-Model - Response to OMG's Request for Proposals on Object Analysis and Design facility [Aagedal et al., 1997a] – member of the SINTEF team in the joint submission working group
- Enterprise Language in ODP - work in ISO JTC1/SC7/WG15 – Norwegian representative
- "Describing Virtual Enterprises: the Object of Roles and the Role of Objects" – OOPSLA 98 workshop [Wood et al., 1998] – co-author
- "Modelling Virtual Enterprises and the Character of Their Interactions" – RIDE-VE99 [Aagedal et al., 1999] – co-author
- "ODP Enterprise Language: UML Perspective" – EDOC '99 [Aagedal and Milosevic, 1999]– co-author and presenter (included in appendix I.A)

7.1.2 Modelling of QoS

- "ODP-based QoS-support in UML" – EDOC'97 – [Aagedal and Berre, 1997] – main author and presenter (included in appendix I.C)
- "Enterprise Modelling and QoS for Command and Control Systems" – EDOC'98 – [Aagedal and Milosevic, 1998] – co-author (included in appendix I.B)
- "Towards an ODP-compliant Object Definition Language with QoS-support" – IDMS'98 – [Aagedal, 1998] – only author and presenter (included in appendix I.D)
- QoS in ODP - work in ISO JTC1/SC7/WG5 – Norwegian representative

- CQML – Technical report [Aagedal, 1999] - an early version of Chapter 4, put forward at ISO JTC1/SC7/WG5 QoS in ODP working meeting in Paris, October, 1999 - only author
- "A Requirements Model for a Quality of Service-aware Multimedia Lecture on Demand System" – to appear in EDMEDIA 2001 [Ecklund et al., 2001b] – co-author
- UML QoS profile – to be submitted to OMG as part of project work in the EU IST project COMBINE

7.1.3 General Computational Support

- "DEM - A Data Exchange Facility for Virtual Enterprises" - WETICE'97 [Aagedal and Oldevik, 1997] – co-author
- Supervisor for Vidar Berget in his work for MSc. Thesis title: "A survey of extended transaction models in a distributed cooperative environment", [Berget, 1998]
- "ODP-modelling of Virtual Enterprises with Supporting Engineering Architecture" – EDOC '99 [Oldevik and Aagedal, 1999] – co-author

7.1.4 Computational QoS-support

- Supervisor for Petter Schau in his work for MSc. Thesis title: "An infrastructure for mobile agents in a distributed object environment", [Schau, 1998].
- Co-supervisor for Brian Elvesæter in his work for MSc. Thesis title: "A Replication Framework Architecture for Mobile Information Systems", [Elvesæter, 2000].
- Supervisor for Tom Christian Paulsen in his work for MSc. Preliminary thesis title: "Runtime Representation of QoS Specifications and Their Semantics" [Paulsen, 2001]. Scheduled for completion May 2001.
- "QoS Management Middleware - A Separable, Reusable Solution" – submitted to IWQoS 2001 [Ecklund et al., 2001a] – co-author
- Supervisor for Lars Sveen Lundby in his work for MSc. Preliminary thesis title: "Specification and Partial Implementation of a QoS framework" [Lundby, 2001]. Scheduled for completion August 2001.

7.2 Major Contributions

The major contribution of this thesis is CQML, a lexical modelling language for precise specification of QoS. It enables specification of QoS at different levels of abstraction, and can be integrated with UML using the CQML-based UML profile. The main novelties of CQML lies in its features for precise specification of QoS characteristics that links QoS specification to QoS measurement, and its features for specification of compositions that enable aggregation of QoS characteristics for composites. Precise specification of QoS characteristics, and thereby precise specification of QoS statements and QoS profiles, is achieved by defining a domain for the QoS characteristics that is used for OCL expressions that specify relations between properties of the modelling elements. An underlying history-based computational model provides additional properties to use in the OCL expressions. The facilities for specification of composition is achieved by using OCL expressions for specification of QoS characteristic values, and by showing that the Abadi-Lamport composition theorem can be applied to most CQML QoS specifications when composing components.

An additional novelty of CQML is its ability to support adaptation both statically, by specification of several QoS profiles and transitions between these, and dynamically, by allowing a run-time representation of QoS specifications and providing means for clients of services to define their worth-value to use when evaluating alternative services.

Another contribution of this thesis is the specification of a representative QoS framework architecture. Even if the purpose of this specification was to show that CQML enables specification of sufficient QoS information for the QoS framework to perform QoS management, its hierarchical structure and the identified components have been refined in [Ecklund et al., 2001a] and is used in ongoing research at the University of Oslo.

Finally, the four papers included in appendix I have contributed with a UML representation of ODP enterprise language concepts, positioning of QoS in ODP enterprise models, positioning of QoS in UML, and a precise QoS specification approach, respectively.

7.3 Research Topic Revisited

This section revisits the research topic presented in section 1.2 and discusses how this has been addressed.

7.3.1 Questions Addressed

In section 1.2, we decomposed the overall problem of including QoS-awareness in software development into a number of subproblems that each was defined by a question. Next we list each question and specify how we have addressed it:

1. How can QoS be positioned in general enterprise modelling?

   The paper included in appendix I.B answers this question by identifying those parts of an ODP enterprise specification to which QoS statements can apply, and by proposing guidelines for enterprise modelling that exploit this. In the paper, this is applied to an oil disaster example. This is also applied in the LoD case study where the UML business model represents one kind of ODP enterprise model to which QoS pertains.

2. Based on the assumption that a well-defined enterprise language in RM-ODP is useful for positioning QoS in enterprise models, what is the mapping between the enterprise language in RM-ODP and the UML?

   The paper included in appendix I.A answers this question by suggesting UML constructs to use for each ODP enterprise language concept. In the paper, an oil disaster example shows how this can be applied. This is also applied in the LoD case study where the business model represents one kind of ODP enterprise model specified in UML.

3. How can QoS be positioned in a general software development methodology for distributed and component-based systems?

   The paper included in appendix I.C answers this question by identifying UML constructs that can be used to specify QoS in UML models. In this paper, QoS is mostly specified as constraints on modelling elements, but at a time when OCL was not part of UML. The paper shows how this can be applied by an example of a car computer system. In the light of CQML, this question is addressed in more detail in Chapter 5 where CQML replaces the simple constraints and where the refinement of QoS specifications in an object-oriented software development methodology is shown through the LoD case study.

4. How can QoS be expressed using UML at different levels of abstraction?

   CQML is designed to be able to express QoS at different levels of abstractions. The LoD case study shows that this can be done. In section 5.4, we present a UML profile for QoS reflecting the modelling concepts in CQML. Hence, we have specified how QoS can be positioned in UML. However, we have not addressed the issue of deciding to what kind of UML modelling elements QoS pertains; the UML profile for QoS only defines how specifications of QoS can be written without considering to what kinds of Classifiers such specifications relate. This is a topic for future work.

5. How can QoS be expressed in a lexical, declarative language and positioned together with interface and component definition languages?

   This is the major topic of this thesis, and the answer given is CQML. The appropriateness of CQML is validated by small examples in Chapter 4 and in the LoD case study in section 5.5.

6. What support is needed from the computational platform to enable QoS-awareness in systems running on it?

   By assuming CQML is sufficient to specify the required QoS aspects that need to be supported by a computational platform, Chapter 6 defines a representative QoS framework architecture that uses information specified in CQML to perform QoS management.

7. How can the lexical language be mapped onto a computational platform that supports QoS?

   The representative QoS framework defined in Chapter 6 uses information specified in CQML to perform QoS management. The QoS Repository, which is a principal part of this QoS framework architecture, is designed to store QoS information other parts of the QoS framework need in order to perform their designated QoS management activities. A prototype implementation of the QoS Repository exists, along with a CQML compiler that fills it.

7.3.2 Research Method

As discussed in section 1.2, a commonly acknowledged research method does not exist in the area of software methodologies and engineering. However, variations of the hypothetical-deductive method are widely used. In contrast to traditional natural sciences, a major part of the research in computer science is often to produce the subject of the hypotheses, i.e., to create a set of constructs (a model) that one hypothesise simplify or represent some aspect better than any known model. That creation of the subject of the hypotheses is a major part of computer science research is due to the fact that computer science often studies man-made phenomena, whereas natural sciences studies phenomena that already exist in nature. This means that, in computer science, one may need to create the phenomena that one later can hypothesise about. In this work, we have used this variation of the hypothetical-deductive method as follows.

The main hypothesis of this thesis is that CQML is sufficient to specify any kind of QoS. This is validated in Chapter 5 where it is shown that CQML can be used in different parts of software development. In particular, the LoD case study shows how CQML can be used on different levels of abstraction during software development. That CQML is well-founded is shown in section 4.7 where a formal representation of the concepts of CQML is provided in addition to a definition of substitutability between components with QoS properties specified in CQML.

A secondary hypothesis is that CQML is more suitable to specify QoS that any other language. This is argued by the evaluation of existing approaches in section 4.4 and their comparison to the features of CQML in subsubsection 4.6.8.2.

It is also a hypothesis of this thesis that CQML provides the information needed for QoS management. This is argued in Chapter 6 where a representative QoS framework architecture that uses CQML-specifications during QoS management is presented. The implemented CQML compiler and run-time representation prototype proves that CQML specifications can be mapped into the QoS Repository, one of the principal QoS framework components.

7.4 Critical Remarks

7.4.1 Are the Results Well-Founded?

A claim of this thesis is that CQML is sufficient to specify QoS. In addition to arguments made during the presentation of the concepts of CQML, a justification of this claim is the use of CQML in the LoD case study. In addition, CQML is shown to be useful in as diverse areas as requirements engineering, process assessment and specification of service level agreements. One may argue that the selection of areas in which CQML is applied, is restricted. We believe, however, it is a representative selection that sufficiently justifies our claim since the areas cover most parts of software development. One may also argue that the areas were not investigated to the sufficient level of details in order to reveal all intricacies where problems may occur. However, we believe we have revealed the issues involved since we have performed a case study to the level of detailed design, and have lent ourselves on results from others for the evaluation of CQML in the areas of requirements engineering, process assessment and service level agreements. Presumably, the results we have used present the significant aspects of the areas and thus, the level of detail for evaluation of the suitability of CQML in these areas is sufficient.

Another claim is that CQML is more suitable to specify QoS than any other language. This claim is justified by investigating relevant languages and identifying missing features in these that CQML supports. One may argue that not all relevant languages were evaluated. However, to the best of our knowledge, we chose the most appropriate languages to evaluate and we are not aware of any other relevant language that targets the same problem domain as CQML. One may also argue that the selection of evaluation criteria is not objective and they may favour CQML. The criteria were selected as a collection of all claimed features of the evaluated languages. If we assume language designers praise the useful features of their language in the material that present it, we believe we have listed those features and conclude that the evaluation criteria cover the features claimed to be useful in such languages. However, the criteria used in the evaluation all focus on technical aspects of the language such as expressability, generality, etc. Other language quality measures such as user friendliness, comprehensability, etc. may also be evaluated; this is a topic for future work, see subsection 8.1.1. Finally, one may argue that there is no empirical support for the claim of the suitability of CQML. An empirical study of this can for instance be a study where a number of developers are presented a problem and some are told to use CQML in specification of QoS whereas others are told to use an alternative language. One could then define some metrics such as time to produce a design, complexity of the design measured for instance in terms of the number of QoS profiles, etc., and compare the results for the different languages under investigation. This has not been conducted and is a topic for future work.

Even if we believe we have sufficiently justified our claims, it is not possible to prove they are true. The number of different cases is unlimited, so the identification of a single case that CQML is unsuited for refutes our first claim. Also, there may exist a language that was not evaluated, that evaluates equally well as CQML; this would refute our second claim. We have, however, done what we believe is necessary to avoid that this would occur.

7.4.2 Are the Results General?

Unless the results are general, they are of little use outside the context in which they were produced. In contrast to, for instance, consultancy and manufacturing, research seeks to produce general results. An appropriate question to ask is therefore whether the results are general or not.

CQML is designed to be a general-purpose QoS specification language and is as such not restricted to any specific QoS category; any kind of QoS characteristic can be defined and combined into QoS profiles for any kind of Classifier. Furthermore, in Chapter 5, CQML was used in different areas of software development covering important phases of the software life-cycle. The diversity of these areas in which CQML was successfully applied indicates that CQML is useful for many kinds of software development activities. Finally, in Chapter 6, CQML was positioned for use in QoS management, showing that CQML is useful in a representative QoS framework architecture. We therefore conclude that CQML is general.

7.4.3 Are the Results Useful?

Assuming the results presented herein are general and well-founded, one may still question their relevance. From the point of view of basic research, this question is often not important since its focus is to produce new knowledge irrespective of its potential use. However, for results in the area of software methodology and engineering that allegedly are meant to support software development, the usefulness of the results is important.

This thesis focuses on QoS. There are those arguing that the concept of QoS will become irrelevant as, using Moore's law, an abundance of resources soon will become available. Assuming this, it follows that the results presented herein are not useful since they all relate to the soon-to-be-outdated concept of QoS. A premise of this conclusion is that QoS is only related to resource scarcity. However, there are QoS categories such as reliability, security, etc., in which faster or more resources not will solve all problems, and hence, QoS will not become irrelevant even if infinitely resource capacities become available. Also, the window of resource scarcity is predicted to extend almost indefinitely with the advent of demanding applications such as distributed virtual reality and real-time stream manipulation [Coulouris et al., 2001] (page 612). Hence, even if QoS were to relate only to resource capacities, QoS is predicted to still be relevant for a long time to come. This view is supported by the strong interest in QoS both in the research community, as shown for instance by the International Workshop on QoS series (IWQoS), and in the commercial world, as shown for instance by the ongoing work in OMG on QoS support in CORBA and UML. We therefore conclude that to focus on QoS does not preclude the results from being useful.

Another argument against the usefulness of the results is that they are not supported by a complete implementation. The QoS framework architecture presented in Chapter 6 has not been implemented, so even if a CQML compiler and a rudimentary QoS repository exist, they are not proven useful in a complete operational QoS framework. However, even if a complete implementation does not exist, we argue that it is feasible to produce one. Other QoS frameworks already exist that provide QoS management facilities, and the QoS framework architecture in Chapter 6 does not differ substantially from them. Furthermore, the history-based computational model for QoS specification that is used by CQML is a widely used model, and the implementation of such a model in a distributed environment can be achieved by appropriate instrumentation of distributed method invocations. Assuming an implementation is produced that supports the necessary features required for QoS management based on QoS specification in CQML, one might nevertheless question the efficiency of such an implementation. In particular, the history information such an implementation needs to provide may become voluminous, so an efficient representation of such histories is needed. A clever implementation of the QoS framework architecture may only record histories used in QoS specifications, and only with a frequency needed for QoS management. We therefore claim that such an implementation can be made, supporting the usefulness of the results. How to implement this efficiently, however, is a topic for future work.

Finally, one may question whether QoS specification is needed for applications for which QoS is relevant. If we assume that QoS is relevant to consider for an application, QoS management needs to be performed in order to monitor and control QoS. QoS management needs QoS specifications in order to know about the QoS characteristics to monitor and control. We can therefore conclude that QoS specification is useful for QoS-aware applications.

Chapter 8 Future Work

This chapter suggests directions for future work.

8.1 Modelling

In this section, we identify possible areas for future work with respect to CQML and its use in modelling.

8.1.1 CQML Quality Evaluation

In the areas of linguistics and semiotics[9], frameworks exist to evaluate the quality of languages. Based on results from this area of research, Krogstie, Sindre, and Lindland [Krogstie et al., 1995, Krogstie, 1995] have developed a framework for discussing the quality of models in general, of which an important part is the quality of the modelling language used to express the models. In [Krogstie, 2001], the author uses this framework to evaluate UML. As a topic for future work, this framework can be used to evaluate the quality of CQML.

The following claims can be used as a starting point for such an evaluation:

- The well-defined grammar supports syntactic correctness.
- CQML is not tied to any specific QoS category, but uses general concepts to define the individual QoS concepts. This suggests domain appropriateness, i.e., whether all statements in the domain can be expressed in the language.
- CQML uses well-established terms such as QoS characteristic and QoS statement; this suggests participant language knowledge appropriateness, i.e., whether the conceptual basis correspond to how the participant (e.g., modeller) perceive reality.
- By being based on well-established terms to specify QoS, CQML also supports knowledge externalisability appropriateness, i.e., whether the participants are able to express their relevant knowledge using the language. In addition, CQML does not force initially complete specifications of QoS. Incomplete specifications can be refined during the development process, allowing participants to express their relevant knowledge despite it being incomplete.
- The comprehensability of CQML strongly relates to the comprehensability of OCL since OCL is used extensively in CQML specifications. OCL was designed to be easily read and written by all practioners of object technology, but it is our experience that many modellers find it hard to use. If the modellers need to use the parts of CQML that use OCL, it may be argued that it does not support comprehensibility appropriateness, i.e., whether all statements in the language are understood by the participants. However, if the increased precision that OCL provides is of no concern, CQML can be used without including OCL specifications, thus provide better comprehensibility appropriateness.
- CQML has a well-defined syntax and semantics, and a mapping to the run-time use of the models. This suggests technical actor interpretation appropriateness, i.e., whether the language lends itself to automatic reasoning.

8.1.2 CQML

More experience with the use of CQML would be useful in order to get a better understanding of the suitability of CQML for QoS specification. Case studies from different domains in which QoS is relevant would be useful to carry out, both to strengthen the justification of the

[9] The study of signs

suitability of CQML, and to possibly identify issues that are not supported by CQML and that need further attention.

A specific issue already identified as an extension to CQML regards compound QoS profiles. Transitions between QoS profiles can currently only be specified using a linear ordering of the QoS profiles specified (in the `precedence`-clause). An extension to this is to allow profile transitions to depend on the part of the profile that was violated; this can for instance be achieved by including a state-based transition scheme.

Definition of QoS characteristic units is another area for CQML extension. In the current version, unit specifications are not treated formally. A formal unit definition would cater for unit conversions. Furthermore, limit values of qualified best-effort QoS statements can only be specified as simple values. To extend this to allow complex qualified best-effort QoS statements is a topic for future work.

CQML supports QoS specification for components providing some service. Specification of QoS policies that embrace the behaviour of many systems is not supported by CQML. Whether to integrate this in CQML or to have a separate policy definition language is an open issue that needs further investigation, as is the definition of the policy definition features themselves.

Empirical studies to investigate the usability of CQML and to compare CQML with other languages could be carried out. In subsection 7.4.1 we outlined some designs of such studies.

8.1.3 Methodology

Even if CQML is useful to use to specify QoS during software development, it is still an open issue when in the software development process QoS specification should be included, and what kind of QoS specification that is useful. As argued in section 5.5, we do not believe there is such a thing as a "right" software development methodology. Nevertheless, we believe guidelines can be developed for practitioners to use when developing software. To develop and justify such guidelines is an area for future research. One way to do this is to perform more case studies to gain experience and from which some general guidelines can be extracted.

Based on the UML profile defined in section 5.4, standard QoS modelling elements can be defined that capture commonly used semantics reusable in many models. For instance, QoS characteristics such as delay, throughput, jitter, etc. could be defined related to a generic model of component interaction. Common design patterns can be used as starting points for such generic models, and the proposed QoS characteristics of general importance in the ISO QoS Framework [ISO/IEC JTC1/SC21, 1995b] can be a useful starting point for standard QoS modelling elements. To specify such standard QoS modelling elements and argue their generality is an area for future work.

In addition to a precise definition in CQML of standard QoS modelling elements, one could also define QoS parameter exchange formats for such standard modelling elements. For instance, to develop an XML DTD suited for conveyance of QoS parameters is a possible topic for future work.

In addition to standard QoS modelling elements, standard QoS-aware functional modelling elements can also be defined. For instance, a stream-stereotype could include a number of QoS characteristics such as delay, throughput and jitter. Such a modelling element could possibly be defined with some standard values to indicate capacity constraints. For instance, an ISDN-stream modelling element could include QoS characteristics such as delay and jitter, and with a capacity constraint on throughput of 128kbit/s.

Finally, the UML profile defined in section 5.4 does not include any directions for a graphical notation. One possibility is to define QoS characteristics as tags that can be exploited in tagged values on the relevant modelling elements, but it would still need further investigation to define a graphical notation for QoS profiles.

8.1.4 Tool Support

Tool support is useful to support developers during QoS specification. The UML profile defined in section 5.4 can be included in an extendible UML modelling tool such as Objecteering [Softeam, 2000] with its UML Profile Builder tool. Such tool support should both enable modellers to use the UML profile for QoS and preferably provide code generation facilities to create CQML that in turn the CQML compiler can use. To develop such tool support is a topic for future work.

A developer may create CQML specifications that have conflicting constraints on parts of the system. To design and develop a tool that is able to identify that such conflicts exist and suggest possible conflict resolution strategies is also a topic for future work.

8.2 Computational Support

Specification of QoS is especially useful if there exists computational support that is able to provide QoS management according to the specifications. The QoS framework architecture presented in Chapter 6 that uses CQML specifications when performing QoS management has not been implemented, but work is ongoing to detail the design and implement parts of it. Several master thesis projects are underway that address parts of the QoS framework architecture, but co-ordination and further efforts are still needed.

In particular, investigation of the use of results from the field of constraint satisfaction in the matching of QoS profiles is an area for future work.

The rudimentary implementation of the QoS run-time repository that stores CQML specifications and makes these available at run-time can be improved. For instance, the repository can be improved to integrate with a model repository containing the functional models, coupling the CQML specifications to the modelling elements they depend on. Such repository integration can for instance be based on a MOF-compliant repository[10], allowing the CQML repository to integrate with any MOF-based modelling tool. In addition, the CQML compiler can be improved to become compliant with the version of CQML presented herein. This would include extensions to handle compositions, QoS profiles and adaptation facilities such as worth functions. The QoS run-time repository can also be improved to support run-time evaluation of substitutability as specified in subsection 4.7.2, the validity of QoS profiles as specified in subsection 4.7.3, relaxed conformance as specified in subsection 4.7.4, and composition of QoS profiles as specified in subsection 4.7.5.

[10] MOF – Meta Object Facility, a standard facility for storing models [OMG, 2000a]

Appendices

I. Papers

This appendix contains four papers as published in their respective proceedings.

I.A Paper 1: ODP Enterprise Language: UML Perspective

This paper discusses the ODP enterprise language and how UML can be used to model the concepts therein.

I.A.1 Novelties and Issues Raised

This paper contributes in two areas: it proposes extensions of new or clarifications of existing ODP enterprise language concepts, and it discusses how concepts in the ODP enterprise language can be modelled using UML. For the ODP enterprise language, the following contributions are made:

- It is argued that the concept of "domain" in some cases may be used instead of the concept of "community", in particular to describe systems such as legal systems;
- Definitions of "objective", "authority" and "ownership" are proposed, and the use of these concepts discussed;
- The need for non-behavioural policy statements, such as those describing QoS, is stated;
- The need for the concept of "process" is discussed.

For the modelling of ODP enterprise language in UML, it is proposed to use:

- Process models, sometimes including swim-lanes or referring to state diagrams, to model most cases of "process" in ODP enterprise language;
- Use case diagrams to model external aspects of a community;
- Class and object diagrams to illustrate static aspects of a community;
- Collaboration diagrams to model dynamic aspects of a community;
- Collaborations to model community templates;
- OCL to partly model policies. However, this restricts policies to deterministic behaviour, excluding modelling of violations from ill-behaved actors;
- Some business rules can be modelled as invariants using OCL.

The major unresolved issue is how to describe policies.

I.A.2 Author Contribution and Evaluation Process

Individual author contribution cannot be identified as the research reported in this paper is a result of joint work during and after a visit at the Distributed Technology Centre (DSTC) in Brisbane, Australia from February to August 1998. Both authors contributed equally to the research and the preparation of the paper.

The paper was presented at EDOC'99, where 28 out of 55 contributions were accepted. Three reviewers reviewed the paper, and suggestions for improvements were addressed for the final version.

I.A.3 Dissemination

The refinement of the enterprise language in RM-ODP is ongoing and both authors were participating in the working group of the standardisation committee. The issues raised in this paper have therefore been considered in this group. In particular, in the final committee draft [ISO/IEC JTC1/SC7/WG17, 2000], "domain" is positioned as one of three common community types. The definitions of authority and ownership are not included, but the terms "owner" and "authorisation" are included, albeit with different definitions than what is proposed in this paper.

Also, in the ongoing process of identifying a UML profile for enterprise distributed object computing (EDOC), issues raised in this paper have been put forward. In particular, the DSTC response [DSTC, 1999] and the Open-IT response (supported by SINTEF) [Open-IT, 1999] to the initial request for proposals include many of the issues put forward in this paper. Currently, work to combine the proposals is under way.

I.B Paper 2: Enterprise Modelling and QoS for Command and Control Systems

This paper discusses QoS in enterprise modelling using an example from the command and control domain.

I.B.1 Novelties and Issues Raised

This paper contributes to positioning of QoS in ODP enterprise modelling. In particular, it proposes a way to position the different types of QoS statement from the ISO QoS framework into the ODP enterprise modelling language.

I.B.2 Author Contribution and Evaluation Process

Individual author contribution cannot be identified as the research reported in this paper is a result of joint work during a visit at the Distributed Technology Centre (DSTC) in Brisbane, Australia from February to August 1998. Both authors contributed equally to the research and the preparation of the paper.

The paper was presented at EDOC '98, where 36 of 65 contributions were accepted. There were three reviewers of the paper, and suggestions for improvements were addressed for the final version.

I.B.3 Dissemination

This paper was put forward as a joint Australian and Norwegian contribution on the working group meeting in Brisbane, 1998, for the QoS in ODP working group. Parts of the text in this paper were then included and do now exist in the draft version of the QoS in ODP Committee Draft version of the standard.

I.C Paper 3: ODP-based QoS-support in UML

This paper discusses modelling of QoS in UML using an example of a car computer system.

I.C.1 Novelties and Issues Raised

This paper contributes to positioning of QoS in UML. In particular, it proposes extensions to UML to be able to model streams and to precisely represent constraints between modelling elements. It structures the models wherein QoS is positioned according to the ODP viewpoints, thereby also presenting an example of positioning QoS in ODP.

I.C.2 Author Contribution and Evaluation Process

The ideas reported in this paper resulted from joint work and discussions. The main author prepared and presented the paper.

The paper was presented at EDOC '97, where 35 of 83 contributions were accepted. There were three reviewers of the paper, and suggestions for improvements were addressed for the final version.

I.C.3 Dissemination

The paper has been used as input to research on modelling distributed systems at SINTEF that has been disseminated into European research projects.

I.D Paper 4: Towards an ODP-compliant Object Definition Language with QoS-support

This paper proposes an approach to precise specification of QoS.

I.D.1 Novelties and Issues Raised

This paper introduces ODL-Q, a language that uses OCL for precise QoS specification based on an ODP-compliant computational model.

ODL-Q has later been refined and improved into CQML as presented in Chapter 4. Both ODL-Q and CQML are languages for precise QoS specification that use OCL and are based on an ODP-compliant computational model, but CQML improves ODL-Q in a number of important ways:

- In ODL-Q, QoS specifications are tightly coupled with TINA ODL. In CQML, on the other hand, QoS specification is de-coupled from component definition.
- QoS constraints are specified as pre- and postconditions on operations in ODL-Q, while in CQML, QoS specification is separated into QoS profiles, QoS statements and QoS characteristics.
- Different kinds of QoS statements (guaranteed, threshold best-effort, compulsory best-effort) can be specified in CQML. Also, derived and specialised QoS characteristics can be specified in CQML.
- CQML adds facilities for composition.
- CQML supports adaptation by compound QoS profiles and worth-functions.

I.D.2 Author Contribution and Evaluation Process

The paper has a single author.

The paper was presented at IDMS '98, where 23 of 68 contributions were accepted. There were three reviewers of the paper, and suggestions for improvements were addressed for the final version.

I.D.3 Dissemination

The results reported in this paper have been further developed into CQML as presented in Chapter 4, and has been put forward in the work on QoS in ODP as an approach for QoS specification.

II. Bibliography

[Abadi, M. and Lamport, L., 1990], *Composing Specifications*, Digital Systems Research Center, Report: 66, Palo Alto, pp. 65.

[Abadi, M. and Lamport, L., 1993], *Conjoining Specifications*, Digital Systems Research Center, Report: 118, Palo Alto, pp. 65.

[Abadi, M. and Lamport, L., 1994] An old-fashioned recipe for real-time, *ACM Transactions on Programming Languages and Systems,* Vol. **16,** 5 (September 1994), pp. 1543-1571.

[ACM, 1999], ACM Digital Library, http://www.acm.org/dl/, 1999.

[Alpern, B. and Schneider, F. B., 1985] Defining liveness, *Information Processing Letters,* Vol. **21,** 4, pp. 181-185.

[Atkinson, C., 1997] *Meta-Modeling for Distributed Object Environments,* Proceedings of First International Enterprise Distributed Object Computing Workshop (EDOC '97), Gold Coast, Australia, pp. 90-101.

[Atkinson, C. and Kühne, T., 2000] *Strict Profiles: Why and How,* Proceedings of Third International Conference on the Unified Modeling Language (UML2000), University of York, UK, pp. 309-322.

[Aurrecoechea, C., Campell, A. and Hauw, L., 1997], *Survey of QoS Architectures*, Center for Telecommunication Research, Columbia University, Report: MPG-95-18.

[Beck, K., 1999] *Extreme Programming Explained: Embrace Change,* Addison-Wesley.

[Becker, C. and Geihs, K., 1997] *MAQS - Management for Adaptive QoS-enabled Services,* Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, San Fransisco, USA.

[Becker, C. and Geihs, K., 1998] *Quality of Service --- Aspects of Distributed Programs,* Proceedings of International Workshop on Aspect-Oriented Programming at ICSE'98, Kyoto, Japan.

[Becker, C. and Geihs, K., 1999] *Generic QoS Specifications for CORBA,* Proceedings of 11. ITG/VDE Fachtagung Kommunikation in Verteilten Systemen (KIVS'99), Darmstadt, Germany.

[Becker, C., Geihs, K. and Gramberg, J., 1999] *Representation of Quality of Service preferences by Contract Hierarchies,* Proceedings of Elektronische Dienstleistungswirtschaft und Financial Engineering (FAN'99), Augsburg, Germany.

[Benington, H. D., 1956] *Production of Large Computer Programs,* Proceedings of ONR Symposium on Advanced Program Methods for Digital Computers, pp. 15-27.

[Berget, V., 1998] A survey of extended transaction models in a distributed cooperative environment, MSc thesis, *Department of Informatics,* University of Oslo, Oslo, pp. 121.

[Berre, A.-J. and Aagedal, J. Ø., 1996] *Role-based Modeling for the ODP viewpoints*, OOPSLA '96 Workshop on Methodologies for Distributed Objects, October 6, 1996.

[Berre, A.-J., Aagedal, J. Ø. and Silva, A. R., 1997] *SIMOD - An ODP-extended Role-Modeling Methodology for Distributed Objects,* Proceedings of Thirtieth Annual Hawaii International Conference on System Sciences, Wailea, Hawaii, pp. 14-23.

[Berry, G. and Gonthier, G., 1988], *The ESTEREL synchronous programming language: design, semantics, implementation*, INRIA, Report: 842.

[Birrel, N. D. and Ould, M. A., 1985] *A Practical Handbook for Software Development,* Cambridge Press, Cambridge.

[Blair, G., Blair, L., Bowman, H., Bryans, J., Chetwynd, A., Derrick, J. and Hutchison, D., 2000] *V-QoS*, http://www.comp.lancs.ac.uk/computing/users/lb/v-qos.html, Accessed: June 11, 2000, Lancaster University and University of Kent, Canterbury.

[Blair, G., Blair, L., Bowman, H. and Chetwynd, A., 1993], *Formal Support for the Specification and Construction of Distributed Multimedia Systems (The Tempo Project): Final Project Deliverable*, Lancaster University, Report: MPG-93-23, pp. 213.

[Blair, G., Blair, L. and Stefani, J.-B., 1997] A Specification Architecture for Multimedia Systems in Open Distributed Processing, *Computer Networks and ISDN Systems,* Vol. **29,** Special Issue on Specification Architecture, pp. 473-500.

[Blair, G. and Stefani, J.-B., 1997] *Open Distributed Processing and Multimedia,* Addison-Wesley.

[Blum, B. I., 1994] A Taxonomy of Software Development Methods, *Communication of the ACM,* Vol. **37,** 11, pp. 82-94.

[Boehm, B. W., 1988] A Spiral Model of Software Development and Enhancement, *IEEE Computer,* Vol. **21,** 5, pp. 61-72.

[Bowman, H., Derrick, J., Linington, P. and Steen, M., 1995] FDTs for ODP, *Computer and Standards Interface,* Vol. **17**, pp. 457-479.

[Brinksma, E., Scollo, G. and Stenbergen, C., 1987] *LOTOS specifications, their implementations and their tests,* Proceedings of IFIP Workshop on Protocol Specification, Testing and Verification VI (PSTV'86), pp. 349-360.

[Chalmers, D. and Sloman, M., 1999] *A Survey of Quality of Service in Mobile Computing Environments*, http://happy.comsoc.org/pubs/surveys/2q99issue/pdf/Sloman.pdf, Accessed: February 29, 1999, IEEE Communications Surveys.

[Chandy, K. M. and Misra, J., 1988] *Parallel Program Design,* Addison-Wesley.

[Chatterjee, S., Sabata, B. and Sydir, J. J., 1998], *ERDoS QoS Architecture*, SRI International, Report: ITAD-1667-TR-98-075, Manlo Park, CA, pp. 54.

[Chung, L., Nixon, B. A. and Yu, E., 1994] *Using Quality Requirements to Systematically Develop Quality Software,* Proceedings of 4th International Conference on Software Quality, McLean, VA, USA.

[Clark, T., Evans, A., Kent, S., Brodsky, S. and Cook, S., 2000], *A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach*, The precise UML group, Report: Version 1.0, pp. 97.

[Clarke, E. M. and Wing, J. M., 1996], *Formal Methods: State of the Art and Future Directions*, Carnegie Mellon University (CMU), Report: CMU-CS-96-178, Pittsburgh, pp. 22.

[Cockburn, A., 1997] Goals and Use Cases, *Journal of Object-Oriented Programming,* Vol. **10,** 5, pp. 35-40.

[Coulouris, G., Dollimore, J. and Kindberg, T., 2001] *Distributed Systems - Concepts and Design,* Addison-Wesley.

[Coulson, G., Blair, G. S., Stefani, J. B., Horn, F. and Hazard, L., 1994], *Supporting the Real-time Requirements of Continuous Media in Open Distributed Processing*, University of Lancaster, Report: MPG-92-35.

[Daniel, J., Traverson, B. and Vignes, S., 1999] *Integration of Quality of Service in Distributed Object Systems,* Proceedings of IFIP TC6 WG6.1 Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99), Helsinki, Finland, pp. 31-43.

[Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J. and Young, P. R., 1989] Computing as a discipline, *Communications of the ACM,* Vol. **32,** 1, pp. 9-23.

[Dietrich, F. and Hubaux, J.-P., 1999], *Formal Methods for Communication Services: Meeting the Industry Expectations*, Institute for Computer Communications and Applications (ICA), Swiss Federal Institute of Technology, Report: TR0002, Lausanne, pp. 23.

[Dijkstra, E. W., 1968] A constructive approach to the problem of program correctness, *BIT,* Vol. **8**, pp. 174-186.

[D'Souza, D. F., Sane, A. and Birchenough, A., 1999] *First Class Extensibility for UML - Packaging of Profiles, Stereotypes, Patterns,* Proceedings of Second International Conference on the Unified Modeling Language: UML'99, Fort Collins, CO, USA, pp. 265-277.

[DSTC, 1999], *DSTC Initial UML Profile for EDOC RFP Submission*, OMG, Report: ad/99-10-07, pp. 92.

[Dykman, N., Griss, M. and Kessler, R., 1999] *Nine Suggestions for Improving UML Extensibility,* Proceedings of Second International Conference on the Unified Modeling Language: UML'99, Fort Collins, CO, USA, pp. 236,248.

[Ecklund, D., Goebel, V., Plagemann, T., Ecklund, E., Griwodz, C., Aagedal, J. Ø., Lund, K. and Berre, A.-J., 2001a] *QoS Management Middleware - A Separable, Reusable Solution,* Proceedings of IWQoS 2001, Karlsruhe, Germany, pp. submitted.

[Ecklund, E. F., Goebel, V. H., Aagedal, J. Ø., Bach-Gansmo, E. and Plagemann, T., 2001b] *A Requirements Model for a Quality of Service-aware Multimedia Lecture on Demand System,* Proceedings of ED-MEDIA 2001 World Conference on Educational Multimedia, Hypermedia & Telecommunications, Tampere, Finland, pp. 2.

[Ehrig, H. and Mahr, B., 1985] *Fundamentals of Algebraic Specification 1,* Springer-Verlag.

[Eliassen, F. and Mehus, S., 1998] *Type Checking Stream Flow Endpoints,* Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), The Lake District, UK.

[Eliassen, F. and Rafaelsen, H. O., 1999] *A Trading Model of Stream Binding Selection,* Proceedings of The Fifth IFIP Conference on Intelligence in Networks (SmartNet'99), Bangkok, Thailand, pp. 251-264.

[Ellsberger, J., Hogrefe, D. and Sarma, A., 1997] *SDL - Formal Object-oriented Language for Communicating Systems,* Prentice Hall Europe.

[Elvesæter, B., 2000] A Replication Framework Architecture for Mobile Information Systems, MSc thesis, *Department of Informatics,* University of Oslo, Oslo, pp. 121.

[Engel, C. and Steinmetz, R., 1993], *Human Perception of Media Synchronization*, IBM European Networking Center, Report: 43.9310, Heidelberg, Germany.

[Fenton, N., 1991] *Software Metrics: A Rigorous Approach,* Chapman-Hall.

[Février, A., Najm, E. and Stefani , J.-B., 1997] *Contracts for ODP,* Proceedings of Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software. Towards a mathematical transformation-based development., Ciudad de Mallorca, Mallorca, Spain.

[Fitzpatrick, T., Blair, G., Coulson, G., Davies, N. and Robin, P., 1998] *Supporting Adaptive Multimedia Applications through Open Bindings,* Proceedings of 4th International Conference on Configurable Distributed Systems (ICCDS '98), Annapolis, Maryland, USA.

[Fluckiger, F., 1995] : Understanding Networked Multimedia(Ed, ITU) Prentice Hall, , pp. 338.

[Frølund, S. and Koistinen, J., 1998a], *QML: A Language for Quality of Service Specification*, Software Technology Laboratory, Hewlett-Packard Company, Report: HPL-98-10, pp. 63.

[Frølund, S. and Koistinen, J., 1998b], *Quality of Service Aware Distributed Object Systems*, Software Technology Laboratory, Hewlett-Packard Company, Report: HPL-98-142, pp. 15.

[Frølund, S. and Koistinen, J., 1998c] Quality-of-Service Specification in Distributed Object Systems, *Distributed Systems Engineering Journal,* Vol. **5,** 4, pp. 179-202.

[Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995] *Design Patterns - Elements of Reusable Object-Oriented Software,* Addison-Wesley.

[Gerhart, S., Craigen, D. and Ralston, T., 1994] Experience with Formal Methods in Critical Systems, In *IEEE Software*, Vol. 11 , January, 1994, pp. 21-28.

[Goebel, V., Eini, I., Lund, K. and Plagemann, T., 1999] *Design, Implementation, and Evaluation of TOOMM: A Temporal Object-Oriented Multimedia Data Model,* Proceedings of 8th IFIP 2.6 Working Conference on Database Semantics (DS-8), Rotorua, New Zealand, pp. 145-168.

[Goldstine, H. H., 1947], *Planning and coding of problems for an electronic computing instrument*, U.S. Army Ord. Dept., Report: Part II, Vol. 1.

[Gross, D. and Yu, E., 2000] *From Non-Functional Requirements to Design through Patterns,* Proceedings of Sixth International Workshop on Requirements Engineering: Foundation for Software Quality, Stockholm, Sweden.

[Guttag, J. V. and Horning, J. J., 1993] *Larch: Languages and Tools for Formal Specification,* Springer-Verlag.

[Hadzilacos, V. and Toueg, S., 1994], *A modular approach to the specification and implementation of fault-tolerant broadcasts*, Department of Computer Science, Cornell University, Report: TR94-1425, Ithaca, pp. 84.

[Hafid, A., 1995] *Hierarchical Negotiation for Distributed Multimedia Applications in a Multi-Domain Environment,* Proceedings of Second Workshop on Protocols for Multimedia Systems, Salzburg, Austria, pp. 397-409.

[Hafid, A. and Bochmann, G. v., 1998] Quality-of-Service adaptation in distributed multimedia applications, *Multimedia Systems,* Vol. **6,** 5, pp. 299-315.

[Hafid, A., Bochmann, G. v. and Kercherve, B., 1996] *A Quality of Service Negotiation Procedure for Distributed Multimedia Presentational Applications,* Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing, pp. 330-339.

[Harel, D., 1987] Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming,* Vol. **8**, pp. 231-274.

[Harmsen, F., Brinkkemper, S. and Oei, H., 1994] In *Methods and Associated Tools for the Information Systems Life Cycle*: Situational Method Engineering for Information System Project Approaches(Eds, Verrijn-Stuart, A. A. and Olle, T. W.) Elsevier, Amsterdam, pp. 169-194.

[Heitmeyer, C. and Mandrioli, D., 1996] *Formal Methods for Real-Time Computing,* John Wiley & Sons.

[Hindin, E., 1998] Say what? QoS in English, In *Network World*, August 17, 1998.

[Hoare, C. A. R., 1969] An Axiomatic Approach to Computer Programming, *Communications of the ACM,* Vol. **12**, pp. 576-580.

[Hoare, C. A. R., 1985] *Communicating Sequential Processes,* Prentice-Hall.

[Hoschka, P., 1998] *Synchronized Multimedia Integration Language (SMIL) 1.0 Specification*, http://www.w3.org/TR/REC-smil/, Accessed: June 12, 1998, W3C.

[Høydalsvik, G. M. and Sindre, G., 1993] On the Purpose of Object-Oriented Analysis, *ACM SIGPLAN Notices,* Vol. **28,** 10, pp. 240-255.

[ICODP '97 position paper session, 1997] *Position paper session at International Conference on Open Distributed Processing and Distributed Platforms (ICODP'97)*, 1997.

[IEE, 1999], INSPEC, http://www.iee.org.uk/publish/inspec/, 1999.

[IEEE/IEE, 1999], IEEE/IEE Electronic Library online, http://198.17.75.40/, 1999.

[ISO, 1986], *Quality Vocabulary*, ISO, Report: ISO 8402, pp. 8.

[ISO, 1991], *Guidelines for the application of ISO 9001 to the development, supply and maintenance of software*, ISO, Report: NS-ISO 9000-3, pp. 29.

[ISO, 1993], *Information Technology - Vocabulary - Part 1: Fundamental Terms*, ISO/IEC, Report: ISO/IEC 2382-1, Genevé.

[ISO/IEC JTC1/SC7, 1998], *Information Technology - Software Process Assessment*, ISO/IEC, Report: 15504.

[ISO/IEC JTC1/SC7, 1999a], *Information Technology - Software product quality - Part 1: Quality model*, ISO/IEC, Report: 9126-1, pp. 25.

[ISO/IEC JTC1/SC7, 1999b], *Information Technology - Software product quality - Part 2: External Metrics*, ISO/IEC, Report: 9126-2, pp. 114.

[ISO/IEC JTC1/SC7, 1999c], *Information Technology - Software product quality - Part 3: Internal Metrics*, ISO/IEC, Report: 9126-3, pp. 31.

[ISO/IEC JTC1/SC7, 2000a], *Information Technology - Software product quality - Part 4: Quality In Use Metrics*, ISO/IEC, Report: 9126-4, pp. 71.

[ISO/IEC JTC1/SC7, 2000b], *Unified Modeling Language -- Part 1: Specification*, ISO, Report: PAS 19501-1.

[ISO/IEC JTC1/SC7/WG17, 2000], *FCD 15414: Information Technology - Open Distributed Processing - Reference Model - Enterprise Viewpoint*, ISO, Report: N2359, pp. 41.

[ISO/IEC JTC1/SC21, 1994], *Basic reference model of open distributed processing - part 2: Foundations*, Report: ITU-T X.902 - ISO/IEC 10746-2.

[ISO/IEC JTC1/SC21, 1995a], *Basic reference model of open distributed processing, part 3: Architecture*, Report: ITU-T X.903 - ISO/IEC 10746-3.

[ISO/IEC JTC1/SC21, 1995b], *QoS - Basic Framework*, ISO, Report: ISO/IEC JTC1/SC 21 N9309.

[ISO/IEC JTC1/SC21, 1997], *ODP Trading Function*, Report: ITU-T X.950 - ISO/IEC 13235.

[ISO/IEC JTC1/SC21, 1998], *Working Draft for Open Distributed Processing - Reference Model - Quality of Service*, ISO, Report: ISO/IEC JTC 1/SC 21 N 10979 Ed 6.4, Berlin.

[ISO/IEC JTC1/SC21/WG7, 1997], *Working document on QoS in ODP*, ISO.

[ISO/IEC JTC1/SC29/WG11, 2000], *Overview of the MPEG-4 Standard*, Report: ISO/IEC JTC1/SC29/WG11 N3747.

[ITU-T, 1994], *Terms and definitions related to quality of service and network performance including dependability*, CCITT/ITU, Report: Recommendation E.800 (08/94).

[IWQoS '97 panel, 1997] *QoS for Distributed Object Computing Middleware -- Fact or Fiction?*, May 22, 1997.

[Jacobson, I., Booch, G. and Rumbaugh, J., 1999] *The Unified Software Development Process,* Addison-Wesley.

[Jennings, N., 2000] *Automated Negotiation and Argumentation*, http://www.ecs.soton.ac.uk/~nrj/neg-arg.html, Accessed: 4. September, 2000

[Jones, C. B., 1983] *Specification and design of (parallel) programs,* Proceedings of Information Processing '83 : IFIP 9th World Congress, Paris, France, pp. 321-332.

[Jones, C. B., 1986] *Systematic Software Development Using VDM,* Prentice-Hall.

[Jones, C. B., 1992], *The Search for Tractable Ways of Reasoning about Programs*, Department of Computer Science, University of Manchester, Report: UMCS-92-4-4, Manchester, UK, pp. 41.

[Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. and Irwin, J., 1997] *Aspect-Oriented Programming,* Proceedings of The European Conference on Object-Oriented Programming (ECOOP'97), Finland.

[Kitson, D. H., 1997] *An Emerging International Standard for Software Process Assessment,* Proceedings of Third International Software Engineering Standards Symposium and Forum, Walnut Creek, CA, USA.

[Koistinen, J., 1997], *Dimensions for Reliability Contracts in Distributed Object Systems*, Software Technology Laboratory, Hewlett-Packard Company, Report: HPL-97-119, pp. 35.

[Koistinen, J. and Seetharaman, A., 1998] *Worth-Based Multi-Category Quality-of-Service Negotiation in Distributed Object Infrastructures,* Proceedings of Second International Enterprise Distributed Object Computing Workshop (EDOC '98), San Diego, CA, USA, pp. 239-249.

[Krogstie, J., 1995] Conceptual Modeling for Computerized Information System Support in Organization, PhD thesis, *Department of Computer and Information Science,* The Norwegian University of Science and Technology, Trondheim.

[Krogstie, J., 2001] In *Unified Modeling Language: Systems Analysis, Design and Development Issues*: Using a Semiotic Framework to Evaluate UML for the Development of Models of High Quality(Eds, Siau, K. and Halpin, T.) Idea Group Publishing, .

[Krogstie, J., Lindland, O. I. and Sindre, G., 1995] *Defining Quality Aspects for Conceptual Models,* Proceedings of IFIP8.1 working conference on Information Systems Concepts (ISCO3); Towards a consolidation of views, Marburg, Germany, pp. 216-231.

[Lakas, A., Blair, G. and Chetwynd, A., 1996] *A Formal Approach to the Design of QoS Parameters in Multimedia Systems,* Proceedings of Fourth International IFIP Workshop on Quality of Service - IWQoS'96, Paris, France.

[Lamport, L., 1994] The Temporal Logic of Actions, *ACM Transactions on Programming Languages and Systems,* Vol. **16,** 3 (May 1994), pp. 872-923.

[Leboucher, L. and Najm, E., 1997] *A framework for real-time QoS in distributed systems,* Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Service, San Francisco.

[Lee, J. and Xue, N.-L., 1999] Analysing user Requirements by Use Cases: A Goal-Driven Apporach, In *IEEE Software*, 1999, pp. 92-101.

[Leue, S., 1995] *Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach,* Proceedings of Fifteenth International Symposium on Protocol Specification, Testing, and Verification (PSTV'95), Warsaw, Poland.

[Leydekkers, P. and Gay, V., 1996] *ODP View on Quality of Service for Open Distributed Multimedia Environments,* Proceedings of 4th International IFIP Workshop on QoS (IWQoS '96), Paris, France.

[Liskov, B. H. and Wing, J. M., 1994] A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems,* Vol. **16,** 6, pp. 1811-1841.

[Loyall, J. P., Schantz, R. E., Zinky, J. A. and Bakken, D. E., 1998] *Specifying and Measuring Quality of Service in Distributed Object Systems,* Proceedings of First International Symposium on Object-Oriented Real-time Distributed Computing (ISORC 98), pp. 43-52.

[Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D. and Mann, W., 1995] Specification and Analysis of System Architecture Using Rapide, *IEEE Transactions on Software Engineering, Special Issue on Software Architecture,* Vol. **21,** 4, pp. 336-355.

[Lundby, L. S., 2001] Specification and Partial Implementation of a QoS framework, MSc thesis, *Department of Informatics,* University of Oslo, Oslo.

[Mackworth, A. K., 1977] Consistency in networks of relations, *Artificial Intelligence,* Vol. **8,** 1, pp. 99-118.

[Marshall, C., 2000] *Enterprise Modeling with UML,* Addison Wesley Longman.

[Medvidovic, N. and Taylor, R. N., 1997] *A Framework for Classifying and Comparing Architecture Description Languages,* Proceedings of 6th European Software Engineering Conference, Zürich, Switzerland, pp. 60-76.

[Milner, R., 1989] *Communication and Concurrency,* Prentice-Hall.

[Miriam-Webster, 2000] *Miriam-Webster Online*, http://www.m-w.com/, 2000, Miriam-Webster, Incorporated.

[Mylopoulos, J., Chung, L. and Nixon, B., 1992] Representing and Using Non-Functional Requirements: A process-Oriented Approach, *IEEE Transactions on Software Engineering,* Vol. **18,** 6, pp. 483-497.

[Najm, E. and Stefani, J.-B., 1995] A Formal Semantics for the ODP Computational Model, *Computer Networks and ISDN Systems,* Vol. **27**, pp. 1305-1329.

[Najm, E. and Stefani, J.-B., 1997] *Computational Models for Open Distributed Systems,* Proceedings of Second IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'97, Canterbery, UK.

[Nixon, B. A., 2000] Management of Performance Requirements for Information Systems, *IEEE Transactions on Software Engineering,* Vol. **26,** 12, pp. 1122-1146.

[Oldevik, J. and Berre, A.-J., 1998] *UML-Based methodology for distributed systems,* Proceedings of Second International Enterprise Distributed Object Computing Workshop (EDOC '98), San Diego, CA, USA, pp. 2-13.

[Oldevik, J. and Aagedal, J. Ø., 1999] *ODP-modelling of Virtual Enterprises with Supporting Engineering Architecture,* Proceedings of Third International Enterprise Distributed Object Computing Conference (EDOC '99), Mannheim, Germany.

[OMG, 1996], *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, Object Management Group.

[OMG, 1997], *Trading Object Service Specification*, Object Management Group.

[OMG, 1998], *Control and Management of Audio/Video Streams, v 1.0*, Object Management Group, Report: telecom/98-06-05, pp. 80.

[OMG, 1999a], *CORBA Components - Volume 1*, Object Management Group, Report: orbos/99-07-01.

[OMG, 1999b], *White Paper on the Profile mechanism*, Object Management Group, Report: ad/99-04-07, pp. 13.

[OMG, 2000a], *Meta Object Facility (MOF) Specification*, Object Management Group, Report: formal/00-04-03, pp. 522.

[OMG, 2000b], *Response to the OMG RFP for Schedulability, Performance, and Time*, Object Management Group, Report: ad/2000-08-04.

[Open-IT, 1999], *A UML Profile for Enterprise Distributed Object Computing*, OMG, Report: ad/99-10-19, pp. 60.

[Ostroff, J., 1992] Formal Methods for the Specification and Design of Real-Time Safety Critical Systems, *Journal of Systems and Software,* Vol. **18,** 1.

[Ould, M. A., 1995] *Business Processes: Modelling and Analysis for Re-engineering and Improvement,* John Wiley & Sons.

[Paulk, M. C., 1999] *Analyzing the Conceptual Relationship Between ISO/IEC 15504 (Software Process Assessment) and the Capability Maturity Model for Software,* Proceedings of The Ninth International Conference on Software Quality, Cambridge, MA, USA, pp. 293-303.

[Paulk, M. C., Curtis, B., Chrissis, M. B. and Weber, C. V., 1993] Capability Maturity Model, Version 1.1, *IEEE Software,* Vol. **10,** 4, pp. 18-27.

[Paulsen, T. C., 2001] Runtime Representation of QoS Specifications and Their Semantics, MSc thesis, *Department of Informatics,* University of Oslo, Oslo.

[Plagemann, T., Eliassen, F., Goebel, V., Kristensen, T. and Rafaelsen, H. O., 1999] *Adaptive QoS Aware Binding of Persistent Multimedia Objects,* Proceedings of International Symposium on Distributed Objects and Applications (DOA'99), Edinburgh, Scotland.

[Plagemann, T., Eliassen, F., Hafskjold, B., Kristensen, T., Macdonald, R. H. and Rafaelsen, H. O., 2000] *Flexible and Extensible QoS Management for Adaptable Middleware,* Proceedings of International Workshop on Protocols for Multimedia Systems (PROMS 2000), Cracow, Poland.

[Pnueli, A., 1977] *The Temporal Logic of Programs,* Proceedings of 18th Annual IEEE Symposium Foundations of Computer Science (FOCS 1977), pp. 46-57.

[Popien, C. and Meyer, B., 1994] *A Service Request Description Language,* Proceedings of FORTE'94, Bern, Switzerland, pp. 17-32.

[Puder, A., Markwitz, S., Gudermann, F. and Geihs, K., 1995] *AI-Based Trading in Open Distributed Environments,* Proceedings of International Conference on Open Distributed Processing (ICODP '95), Brisbane, Australia.

[Rajahalme, J., Mota, T., Steegmans, F., Hansen, P. F. and Fonseca, F., 1997] *Quality of Service Negotiation in TINA,* Proceedings of Global Convergence of Telecommunications and Distributed Object Computing, TINA 97, pp. 278-286.

[Reenskaug, T., Wold, P. and Lehne, O. A., 1996] *Working with Objects - The OOram Software Engineering Method,* Manning Publications.

[Ren, S., Venkatasubramanian, N. and Agha, G., 1997] *Formalizing Multimedia QoS Constraints Using Actors,* Proceedings of IFIP TC6 WG6.1 International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '97), Canterbury, Kent, pp. 139-153.

[Rogerson, D., 1997] *Inside COM,* Microsoft Press.

[Rothermel, K., Dermler, G. and Fiederer, W., 1997] *QoS Negotiation and Resource Reservation for Distributed Multimedia Applications,* Proceedings of IEEE International Conference on Multimedia Computing and Systems '97, pp. 319-326.

[Royce, W. W., 1970] *Managing the development of large software systems: concepts and techniques,* Proceedings of WESCON, Los Angeles, USA, pp. 1-9.

[Ruttkay, Z., 1998] Constraint Satisfaction - a Survey, *CWI Quarterly,* Vol. **11,** 2-3, pp. 163-214.

[Römer, K., Puder, A. and Pilhofer, F., 2000], micoORB, http://www.mico.org/, 2000.

[Sabata, B., Chatterjee, S., Davis, M., Sydir, J. J. and Lawrence, T. F., 1997] *Taxonomy for QoS Specifications,* Proceedings of IEEE Computer Society 3rd International Workshop on Object-oriented Real-time Dependable Systems (WORDS '97), Newport Beach, CA, USA.

[Schau, P., 1998] An infrastructure for mobile agents in a distributed object environment, MSc thesis, *Department of Informatics,* University of Oslo, Oslo, pp. 102.

[Schmidt, D. C., 1997] Recent Advances in Distributed Computing, In *IEEE Communications Magazine*, Vol. 14 , February, 1997, pp. 42-44.

[Schmidt, D. C., Levine, D. L. and Mungee, S., 1998] The Design and Performance of the TAO Real-Time Object Request Broker, *Computer Communications,* Vol. **21,** 14, pp. 46.

[Shannon, B., 1999], *Java 2 Platform Enterprise Edition Specification, 1.2.1*, Sun Microsystems, Palo Alto, pp. 140.

[Silva, A. R., Gonçalves, T., Rosa, F., Berre, A.-J. and Aagedal, J. Ø., 1997] *Organization, Information System and Distribution Modelling: An Integrated Approach,* Proceedings of First International Enterprise Distributed Object Computing Workshop (EDOC '97), Gold Coast, Australia, pp. 57-65.

[Sluman, C., Tucker, J., LeBlanc, J. P. and Wood, B., 1997], *Quality of Service (QoS) OMG Green Paper*, OMG, Report: om/97-06-04.

[Softeam, 2000], Objecteering, http://www.softeam.fr/us/pobj_pro.htm, 2000.

[Software Engineering Institute, 2000] *Capability Maturity Model for Software*, http://www.sei.cmu.edu/cmm/cmms/cmms.html, Accessed: 3. April, 2000, Carnegie Mellon University.

[Solberg, A., Neple, T., Oldevik, J. and Kvalheim, B., 1999] *A flexible framework for development of component-based distributed systems,* Proceedings of IFIP TC6 WG6.1 Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99), Helsinki, Finland, pp. 149-162.

[Spivey, J. M., 1988] *Understanding Z: A Specification language and its formal semantics,* Cambridge University Press.

[Stefani, J. B., 1993] *Computational Aspects of QoS in an Object Based Distributed Architecture,* Proceedings of 3rd Int. Workshop on Responsive Computer Systems, Licoln, New Hampshire, USA.

[Sturm, R., Morris, W. and Jander, M., 2000a] *Foundations of Service Level Management,* SAMS.

[Sturm, R., Morris, W. and Jander, M., 2000b] *Sample Internal SLA (short form)*, http://www.nextslm.org/slm_short.html, Accessed: August 7, 2000b

[Stølen, K., 1998], *Formal Specification of Open Distributed Systems - Overview and Evaluation of Existing Methods*, Institutt for Energiteknikk, Report: HWR-523, Halden, pp. 88.

[Sørumgård, S., 1997] Verification of Process Conformance in Empirical Studies of Software Development, PhD thesis, *Department of Computer and Information Science,* The Norwegian University of Science and Technology, Trondheim, pp. 252.

[The Open Group, 1997], *DCE 1.1: Remote Procedure Call*, The Open Group, Report: C706, Reading, England, pp. 737.

[TINA-C, 1994], *Quality of Service Framework*, TINA-C, Report: TP_MRK.001_1.0_94.

[TINA-C, 1996], *TINA Object Definition Language Manual*, TINA-C, Report: TP_NM.002_2.2_96.

[Tsang, E. P. K., 1993] *Foundations of Constraint Satisfaction,* Academic Press.

[Turing, A., 1949] *Checking a large routine,* Proceedings of Report of a Conference on High Speed Automatic Calculating Machines, University Mathematical Laboratory, Cambridge, U, pp. 67-69.

[UML RTF, 1999], *OMG UML v1.3*, Object Management Group, Report: ad/99-06-08.

[Vanegas, R., Zinky, J. A., Loyall, J. P., Schantz, R. E. and Bakken, D. E., 1998] *QuO's Runtime Support for Quality of Service in Distributed Objects,* Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), Lake District, Lancaster, UK, pp. 207-222.

[Vasudevan, V., 1998a] *Augmenting OMG traders to handle service composition*, http://www.objs.com/survey/compositional-trader.html, Accessed: January 7, 1998a, Object Services and Consulting, Inc.

[Vasudevan, V., 1998b] *A Reference Model for Trader-Based Distributed Systems Architectures*, http://www.objs.com/survey/trader-reference-model.html, Accessed: January, 1998b, Object Services and Consulting, Inc.

[Vieru, V., 1998] Vague Concepts and Sorites Paradox, *Studies in Informatics and Control,* Vol. **7,** 3.

[Vogel, A., Kerhervé, B., Bochmann, G. v. and Gecsei, J., 1995] Distributed Multimedia and QoS - A Survey, *IEEE Multimedia,* Vol. **2,** 2, pp. 10-19.

[Waddington, D. G. and Hutchison, D., 1998] *A General Model for QoS Adaptation,* Proceedings of Sixth International Workshop on Quality of Service (IWQoS 98), pp. 275-277.

[Wang, C., Ecklund, E. F., Goebel, V. H. and Plagemann, T., 2001] *Design and Implementation of a LoD System for Multimedia-Supported Learning at the Medical Faculty,* Proceedings of ED-MEDIA 2001 World Conference on Educational Multimedia, Hypermedia & Telecommunications, Tampere, Finland, pp. to appear.

[Warmer, J. B. and Kleppe, A. G., 1999] *The Object Constraint Language: Precise Modeling with UML,* Addison-Wesley, Reading, Mass., USA.

[Wood, A., Milosevic, Z. and Aagedal, J. Ø., 1998] *Describing Virtual Enterprises: the Object of Roles and the Role of Objects*, Objects, Components and the Virtual Enterprise '98 - An OOPSLA '98 Workshop, October 19, 1998.

[Zhang, X., Podorozhny, R. and Lesser, V., 2000], *Cooperative, MultiStep Negotiation Over a Multi-Dimensional Utility Function*, Multi-Agent Systems Laboratory, Department of Computer Science, University of Massachusetts at Amhurst, Report: 2000-02, pp. 6.

[Zinky, J. A., Bakken, D. E. and Schantz, R., 1995] *Overview of Quality of Service for Distributed Objects,* Proceedings of Dual Use Technologies Conference, Utica NY, USA.

[Zinky, J. A., Bakken, D. E. and Schantz, R. E., 1997] Architectural Support for Quality of Service for CORBA Objects, In *Theory and Practice of Object Systems*, Vol. 3 , 1997, 1997.

[Aagedal, J. Ø., 1998] *Towards an ODP-compliant Object Definition Language with QoS-support,* Proceedings of 5th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'98), Oslo, Norway, pp. 183-194.

[Aagedal, J. Ø., 1999], *Component Quality Modelling Language (CQML)*, SINTEF, Report: STF40 A99048, Oslo, pp. 25.

[Aagedal, J. Ø. and Berre, A.-J., 1997] *ODP-based QoS-support in UML,* Proceedings of First International Enterprise Distributed Object Computing Workshop (EDOC '97), Gold Coast, Australia, pp. 310-321.

[Aagedal, J. Ø., Berre, A.-J., Cockburn, A., Oldevik, J., Reenskaug, T. e., Reich, G.-P. and Wirfs-Brock, R., 1997a], *The OOram Meta-Model*, OMG, Report: ad/97-01-15.

[Aagedal, J. Ø., Berre, A.-J., Goebel, V. and Plagemann, T., 1997b], *Object-Oriented Role-Modeling with QoS Support for the ODP Viewpoints*, SINTEF, Report: STF40 A97067.

[Aagedal, J. Ø. and Milosevic, Z., 1998] *Enterprise Modelling and QoS for Command and Control Systems,* Proceedings of Second International Enterprise Distributed Object Computing Workshop (EDOC '98), San Diego, CA, USA, pp. 88-101.

[Aagedal, J. Ø. and Milosevic, Z., 1999] *ODP Enterprise Language: UML Perspective,* Proceedings of Third International Enterprise Distributed Object Computing Conference (EDOC '99), Mannheim, Germany.

[Aagedal, J. Ø., Milosevic, Z. and Wood, A., 1999] *Modelling Virtual Enterprises and the Character of Their Interactions,* Proceedings of Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDE-VE99), Sydney, Australia, pp. 19-26.

[Aagedal, J. Ø. and Oldevik, J., 1997] *DEM: A Data Exchange Facility for Virtual Enterprises,* Proceedings of Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE '97), Cambridge, Massachusetts, USA, pp. 17-22.

[Aagesen, F. A., 1997] QoS frameworks for open distributed processing systems, *Telektronikk,* Vol. **93,** 1, pp. 26-41.

III. CQML Grammar

The grammar for CQML (LL(2)) is defined below. The grammar description uses EBNF syntax where angle-brackets “<” and “>” enclose non-terminals, “[” and “]” enclose an optional part, “|” means choice, “*” means zero or more times, “+” means one or more times and ::= denotes productions. Curly brackets (“{“ and “}”) are used for grouping while single quotes (‘’) are used to enclose terminals. Keywords are shown in **bold**.

We do not repeat the OCL grammar, see [UML RTF, 1999]. Non-terminals prefixed “OCL::” refer to a corresponding non-terminal in the OCL grammar without this prefix.

```
<quality_declaration>          ::=  <cqml_declaration>*
<cqml_declaration>             ::=  <category_declaration>
                               |    <characteristic_declaration>
                               |    <quality_declaration>
                               |    <profile_declaration>

<category_declaration>         ::=  quality_category <category_name>
                                    ‘{’ <category_body> ‘}’
<category_body>                ::=  <cqml_declaration>+
                               |    <cqml_reference>+
<cqml_reference>               ::=  <characteristic_name> ‘;’
                               |    <quality_name> ‘;’
                               |    <profile_name> ‘;’
                               |    <category_name> ‘;’

<characteristic_declaration>   ::=  quality_characteristic <characteristic_name>
                                    [ ‘(’ <OCL::formalParameterList> ‘)’ ]
                                    { ‘{’ [ <individual_body> ] ’}’
                                     | { <parent_characteristic>
                                       ‘{’ [ <specialisation_body> ]‘}’ }
<parent_characteristic>        ::=  ‘:’ <parent_name> [‘(’ <OCL::actualParameterList> ‘)’]
<individual_body>              ::=  <characteristic_definition> | <functional_derivation>
<characteristic_definition>    ::=  <domain_declaration> [<statistical_derivation>]
                                    [<values_declaration>] [<composition_declaration>]
                                    [<invariant>]
<specialisation_body>          ::=  <characteristic_definition> | <specialisation_derivation>

<domain_declaration>           ::=  domain ‘:’ <domain_type> [<unit>] ‘;’
<domain_type>                  ::=  <numeric>
                               |    <set>
                               |    <enumeration>
<numeric>                      ::=  [<direction>] numeric [<type_specification>]
                                    [<boundary_restriction>]
<direction>                    ::=  increasing
                               |    decreasing
<type_specification>           ::=  real
                               |    integer
                               |    natural
<boundary_restriction>         ::=  <lower_boundary> <OCL::number> ‘..’
                                    [ <OCL::number> ] <upper_boundary>
                               |    <lower_boundary> ‘..’
                                    <OCL::number> <upper_boundary>
<lower_boundary>               ::=  ‘[’ | ‘(’
<upper_boundary>              ::=  ‘]’ | ‘)’
<set>                         ::=  <simple_set>
                              |    <direction> <simple_set> [with <order_definition>]
<simple_set>                  ::=  set ‘{’ <name_list> ‘}’
<order_definition>            ::=  order ‘{’ <one_order> {‘,’ <one_order>}* ‘}’
<one_order>                   ::=  <element_name> <order_operator> <element_name>
<order_operator>              ::=  ‘<’ | ‘>’
<enumeration>                 ::=  <simple_enumeration>
                              |    <direction> <simple_enumeration>
                                   [with <order_definition>]
<simple_enumeration>          ::=  enum ‘{’ <name_list> ‘}’
<unit>                        ::=  <name>  [ ‘/’ <name> ]

<values_declaration>          ::=  values ‘:’ <OCL::expression> ‘;’

<composition_declaration>     ::=  composition ‘:’ <composition_definition>
<composition_definition>      ::=  <parallel_and> [<parallel_or>] [<sequential>]
                              |    <parallel_or> [<sequential>]
                              |    <sequential>
<parallel_and>                ::=  parallel-and <binary_expression> ‘;’
<parallel_or>                 ::=  parallel-or <binary_expression> ‘;’
<sequential>                  ::=  sequential <binary_expression> ‘;’
<binary_expression>           ::=  <OCL::multiplicativeExpression>
                              |    <OCL::additiveExpression>
                              |    <method>

<invariant>                   ::=  invariant <OCL::expression> ‘;’

<specialisation_derivation>   ::=  <statistical_derivation> [<invariant>]
                              |    <functional_derivation>
<functional_derivation>       ::=  {<characteristic_name> ‘;’}+
                                   [<characteristic_definition>]
<statistical_derivation>      ::=  {<one_aspect> ‘;’}+
<one_aspect>                  ::=  maximum
                              |    minimum
                              |    range
                              |    mean
                              |    variance
                              |    standard_deviation
                              |    percentile <OCL::number>
                              |    moment <OCL::number>
                              |    frequency {<element> | <range_values>}
                              |    distribution <distribution>
<range_values>                ::=  <lower_boundary> <elements> <upper_boundary>
<elements>                    ::=  <element> ‘..’ <element>
<element>                     ::=  <OCL::name> | <OCL::number>
<distribution>                ::=  Poisson <mean_number>
                              |    <distribution_name> [<distribution_parameter>
                                   {‘,’ <distribution_parameter>}* ]
<distribution_parameter>      ::=  <element>
```

```
<quality_declaration>          ::=  quality <quality_name>
                                    ['(' <OCL::formalParameterList> ')']
                                    [<quality_specialisation>] '{' {<constraint> ';'}*
                                    [<invariant>] '}'
<quality_specialisation>       ::=  ':' <quality_name>
                                    [ '(' <OCL::actualParameterList> ')']
<constraint>                   ::=  <simple_constraint> | <qualified_constraint>
<simple_constraint>            ::=  <single_constraint>
                                    { <OCL::logicalOperator> <simple_constraint> }*
                                |   '(' <simple_constraint> ')'
                                |   <OCL::logicalExpression>
<single_constraint>            ::=  <characteristic_name>
                                    ['(' <OCL::actualParameterList) ')' ]
                                    [ '.' <one_aspect> ] <OCL::relationalOperator> <value>
<qualified_constraint>         ::=  guaranteed <simple_constraint>
                                |   best-effort <simple_constraint>
                                |   <qualifier> best-effort <single_constraint>
                                    with limit <element>
<value>                        ::=  <element>
                                |   <range_values>
                                |   <characteristic_name>
                                    ['(' <OCL::actualParameterList) ')' ][ '.' <one_aspect> ]
<qualifier>                    ::=  threshold | compulsory

<profile_declaration>          ::=  profile <profile_name> [<profile_specialisation>]
                                    for <component_name> '{' [<profile_body>] '}'
<profile_specialisation>       ::=  ':' <profile_name>
<profile_body>                 ::=  <single_profile_body>
                                |   <one_profile>+ <transition>+ [ <precedence> ]
<single_profile_body>          ::=  { <expectation_specification> [ <offer_specification> ]
                                      | <offer_specification> } [<invariant>]
<expectation_specification>    ::=  uses <expectation> ';'
<offer_specification>          ::=  provides <offer> ';'
<expectation>                       <expectation_name>
                                    ['(' <OCL::actualParameterList> ')']
                                    [worth <method>]
                                    {<OCL::logicalOperator> <expectation>}*
                                |   '(' <expectation> ')'
<offer>                        ::=  <offer_name>
                                    ['(' <OCL::actualParameterList> ')']
                                    {<OCL::logicalOperator> <offer> }* ';'
                                |   '(' <offer> ')'
<one_profile>                  ::=  profile <profile_name>
                                    '{' <single_profile_body> '}'
<transition>                   ::=  transition <transition_profiles> ':'
                                    <transition_body>
<transition_profiles>          ::=  {<profile_name> | any } '->' <profile_name>
                                |   <profile_name> '->' any
<transition_body>              ::=  <method> ';'
                                |   '{' {<method> ';'}+ '}'
<precedence>                   ::=  precedence ':' <profile_name>
                                    { ',' <profile_name> }+ ';'

<category_name>                ::=  <name>
```

```
<characteristic_name>       ::=  <name>
<quality_name>              ::=  <name>
<profile_name>              ::=  <name>
<parent_name>               ::=  <name>
<element_name>              ::=  <name>
<distribution_name>         ::=  <name>
<component_name>            ::=  <name>
<expectation_name>          ::=  <quality_name>
<offer_name>                ::=  <quality_name>
<name>                      ::=  <OCL::name>
<name_list>                 ::=  <name> {‘,’ <name>}*
<mean_number>               ::=  <OCL::number>
<method>                    ::=  <OCL::featureCall>
```

IV. IDL for Run-time Representation

The IDL of the QoS repository that stores the run-time representation of CQML is provided below as implemented, see section 4.9.

```
enum relOp { eq, ne, lt, le, gt, ge };

enum DefinitionKind {
    dk_qos_characteristic,
    dk_qos_statement,
    dk_qos_category,
    dk_qos_reference,
    dk_qos_repository
};

typedef string Identifier;
typedef string ScopedName;

typedef sequence<Identifier> IdentifierSeq;

enum DomainKind {
    unordered_enum,
    ordered_incr_enum,
    ordered_decr_enum,
    unordered_set,
    incr_set,
    decr_set,
    ordered_incr_set,
    ordered_decr_set,
    incr_numeric,
    decr_numeric
};

enum ValueKind { vk_number, vk_element, vk_set };

enum AspectKind {
    ak_none,
    ak_minimum,
    ak_maximum,
    ak_range,
    ak_mean,
    ak_variance,
    ak_standard_deviation,
    ak_percentile,
    ak_moment,
    ak_frequency
};

enum ParamKind {
    pk_number,
    pk_range
};

struct Value {
    ValueKind kind;
    float number;
    Identifier element;
    IdentifierSeq vset;
};
```

```
struct ValueOrder {
    Identifier low;
    Identifier high;
};

typedef sequence<ValueOrder> ValueOrderSeq;

exception NameCrash{};

interface qos_characteristic;
interface qos_statement;
interface qos_category;
interface qos_reference;

interface QRObject {
    readonly attribute DefinitionKind defKind;
};

typedef sequence<QRObject> QRObjectSeq;

interface Container;

interface Contained : QRObject {
    readonly attribute Identifier id;
    Container get_container();
};

typedef sequence<Contained> ContainedSeq;

//  container for storing qr things, and looking them up
interface Container : QRObject {

    // make a new qos-object, and add to container
    qos_characteristic create_qos_characteristic( in Identifier id,
                                  in DomainKind dom )
      raises (NameCrash);

    qos_statement create_qos_statement( in Identifier id )
      raises (NameCrash);

    qos_category create_qos_category( in Identifier id )
      raises (NameCrash);

    qos_reference create_qos_reference( in Contained obj )
      raises (NameCrash);

    // see if you can find the qos-object by name
      Contained find_name( in ScopedName name );
    ContainedSeq list();

    void remove(in Contained qo);
};

struct Range {
    float min;
    float max;
    boolean min_include;
    boolean max_include;
};
```

```
struct AspectParam {
   ParamKind kind;
   float number;
   Range range;
};

interface qos_characteristic : Contained {
   // characteristic is a refinement of some other
   void refines(in qos_characteristic base);

   // set the domain elements
   void domain_values(in IdentifierSeq vals);

   // specify order
   void domain_order(in ValueOrderSeq rels);

   // aspects
   void simple_aspect(in AspectKind ak);

   void param_aspect(in AspectKind ak, in AspectParam param);

   // defined?
   boolean simple_aspect_defined(in AspectKind ak);
   boolean param_aspect_defined(in AspectKind ak,
                           in AspectParam param);

   // operation to test if this characteristic has equal semantics
   // to the argument characteristic
   boolean has_equal_semantics(in qos_characteristic qc );

   // is this a value in the domain?
   boolean value_in_domain(in Value val);

   // is the first value stronger than the last?
     short compare(in Value strong, in Value weak );
};

struct AspectDescription {
   AspectKind kind;
   AspectParam param;
   relOp op;
   Value value;
};

typedef sequence<AspectDescription> AspectDescriptionSeq;

struct ConstraintDescription {
   qos_characteristic characteristic;
   AspectDescriptionSeq aspects;
};

typedef sequence<ConstraintDescription> ConstraintDescriptionSeq;

interface qos_statement : Contained {
   // refines this statement
   void refines(in qos_statement base);

   // strenghten the statement with yet another constraint
   void add_constraint(in qos_characteristic qc,
              in AspectDescription aspect);
```

```
    // maybe i am stronger than that argument guy
    boolean conforms_to(in qos_statement qs);

    // return a sequence of constraints
    ConstraintDescriptionSeq constraints();
};

interface qos_category : Container, Contained {
};

interface qos_reference : Contained {
    Contained get_reference();
};

interface QoSRepository : Container {
};
```

V. Index