Abstract

This work shows how the Component Quality Modelling Language (CQML) can be represented in a runtime system, and how it can easily be parsed to generate objects that can be used at runtime.

CQML is a language for modelling the Quality of Service (QoS) of a distributed system. QoS is not about what a service does, but how well it is done.

To be successful, a distributed application must have satisfactory QoS. With traditional modelling tools, we can specify that an object receives a stream of video from some other object. With CQML we can specify that this stream should deliver video at a framerate of at least 25 frames per second.

CQML enables us to define the QoS of a system at design time, and to generate code from this specification that can be used for managing QoS at runtime. This way, we deal with issues related to QoS at an early stage in the development of the system, instead of solving them in an ad hoc fashion afterwards.

To show that CQML is suited for use in a real application, we create a runtime representation of the CQML concepts, and a repository where the objects that represents QoS specifications can live. We then create a parser that generates these objects automatically.
# Contents

1 Introduction .......................................................... 1
   1.1 Motivation ...................................................... 1
   1.2 Problem statement .............................................. 1
   1.3 Method .......................................................... 2
      1.3.1 CORBA .................................................... 2
      1.3.2 GNU C++ .................................................. 2
      1.3.3 Linux ...................................................... 2
      1.3.4 JavaCC .................................................... 2
   1.3.5 Program and error .......................................... 2
   1.4 Progression ..................................................... 3
   1.5 Acknowledgements .............................................. 3
   1.6 Structure ........................................................ 3

2 Theory .................................................................. 5
   2.1 Distributed Objects ............................................. 5
      2.1.1 Distributed systems ......................................... 5
      2.1.2 The client-server model ..................................... 7
      2.1.3 Introducing objects ........................................... 8
      2.1.4 RM-ODP ...................................................... 9
      2.1.5 CORBA ...................................................... 10
      2.1.6 Multimedia / Shortcomings ................................ 11
   2.2 Quality of Service ............................................... 12
   2.3 Object Constraint Language .................................... 13

3 Requirements .......................................................... 14
   3.1 QoS modeling language requirements .............................. 14
   3.2 What we require of the runtime system ........................... 16

4 Related work .......................................................... 18
   4.1 QoS Modeling Language ........................................... 18
      4.1.1 QoS Runtime Representation .............................. 19
   4.2 Quality Objects .................................................... 20
      4.2.1 Connection ................................................... 21
4.2.2 Contract ........................................ 21
4.2.3 System properties .............................. 21
4.2.4 Adapting ....................................... 22
4.2.5 Delegates ...................................... 22
4.2.6 Aspect languages .............................. 22
4.3 ERDoS ........................................... 23
4.3.1 QoS Taxonomy ................................. 23
4.3.2 Logical Application Stream Model .......... 24
4.3.3 Resource Model ............................... 24
4.3.4 System Model .................................. 25
4.4 QMF ............................................. 25
4.5 Summary ......................................... 25

5 CQML .................................................. 26
5.1 QoS characteristic ............................... 26
  5.1.1 Domain ....................................... 27
  5.1.2 Statistical aspects ........................... 27
  5.1.3 Specialization ............................... 28
  5.1.4 Semantics ................................... 28
5.2 QoS category ..................................... 30
5.3 QoS statement .................................... 30
5.4 QoS offers and requirements .................... 31
5.5 QoS contract ..................................... 32

6 Design ............................................. 33
6.1 Architecture ................................... 33
  6.1.1 QosRepository information objects ....... 34
  6.1.2 QoS management components ............... 35
6.2 Repository ..................................... 38
  6.2.1 Simplifications .............................. 38
  6.2.2 QRObj ect .................................. 38
  6.2.3 Container .................................. 40
  6.2.4 Contained .................................. 41
  6.2.5 QoSRepository .............................. 42
  6.2.6 qos_category ................................ 42
  6.2.7 qos_characteristic ......................... 42
  6.2.8 qos_statement ............................... 45
  6.2.9 qos_reference ............................... 47
6.3 QoS Negotiation and Monitoring ................. 48
  6.3.1 Negotiator .................................. 48
  6.3.2 qos_real .................................... 49
  6.3.3 qos_monitor ................................ 52
6.4 CQML Parser ................................... 54
  6.4.1 Object language and symbol resolution ... 55
| 6.4.2 | Symbol table | 56 |
| 6.4.3 | Parse tree | 57 |
| 6.4.4 | Emitter | 61 |
| 6.4.5 | Populating and using the repository | 62 |

### 7 Implementation

| 7.1 | Repository | 63 |
| 7.1.1 | Lifecycle | 63 |
| 7.1.2 | qos_characteristic equality | 63 |
| 7.1.3 | qos_statement conformance | 63 |
| 7.1.4 | CORBA references | 67 |
| 7.1.5 | Not yet implemented | 67 |
| 7.2 | Parser | 68 |
| 7.2.1 | CQML parser | 68 |
| 7.2.2 | CQML and the object language | 69 |
| 7.2.3 | Emitter | 69 |
| 7.3 | Test case | 70 |

### 8 Evaluation

| 8.1 | Repository | 77 |
| 8.1.1 | Concepts | 77 |
| 8.1.2 | Overhead | 77 |
| 8.1.3 | Platform independence | 78 |
| 8.1.4 | Repository interaction | 79 |
| 8.1.5 | Design time vs runtime specifications | 79 |
| 8.1.6 | Anonymous QoS objects | 80 |
| 8.1.7 | Parameters | 80 |
| 8.1.8 | QoS monitoring | 80 |
| 8.2 | CQML and code generation | 80 |
| 8.2.1 | Parsing | 80 |
| 8.2.2 | Generated code | 81 |
| 8.2.3 | Adaptation | 81 |
| 8.2.4 | CQML constructs | 81 |

### 9 Conclusion and Future work

<p>| 9.1 | Conclusion | 82 |
| 9.1.1 | Runtime representation | 82 |
| 9.1.2 | Parser | 83 |
| 9.2 | Future work | 83 |
| 9.2.1 | Improvements to the repository | 83 |
| 9.2.2 | Unimplemented features | 84 |
| 9.2.3 | Parser | 84 |
| 9.2.4 | Tools | 84 |
| 9.2.5 | Benchmarks | 84 |</p>
<table>
<thead>
<tr>
<th></th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>IDL</td>
<td>86</td>
</tr>
<tr>
<td>B</td>
<td>JavaCC code</td>
<td>93</td>
</tr>
<tr>
<td>C</td>
<td>QoS runtime library</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>120</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

We have seen an incredible evolution in technology. Technology is improving continually. Computers are getting faster, network speeds are faster. With new technologies comes new possibilities, but also higher demands.

In traditional system modelling, we have been focusing on the functional aspects. With more demanding applications, we will also need a way to model the behavioural aspects of a system. What Quality of Service (QoS) does an application need to be able to function satisfactorily. CQML is one such language for modelling QoS.

It is not sufficient to be able to model QoS. A distributed system is very dynamic in nature, and resources can become scarce. Some resources can be expensive, so we will have to negotiate the use of them, and also monitor that we get the resources we pay for. Some applications might stop to work, or be useless if they do not get good enough QoS.

If the QoS drops below some level, we might need to take action. This can include trying to connect to some service with better QoS, degrading some less important functions, or even sound an alarm.

We need a way to represent CQML in the runtime system, and use it to manage the QoS of the system.

1.2 Problem statement

Can the concepts of CQML be represented in a runtime system?

In this work we focus on creating a runtime representation of the concepts of CQML that can be used to store QoS specifications, and use them in the management of QoS. The QoS specifications will live in a QoS repository, that we will implement.

Is CQML well suited for parsing and automatic code generation?
We will also focus on creating a parser for CQML that will populate the repository with QoS specifications.
This work will not focus on using the semantics of CQML to check for semantic equality, or generate code used for negotiation and monitoring.

1.3 Method

1.3.1 CORBA
The choice of a framework for distributed applications fell on CORBA, as it is a widely supported framework. Its Interface Definition Language, is also not far from the Object Definition Language that CQML uses, or extends.

The CORBA ORB chosen for this work was the MICO ORB. MICO is a core implementation of a CORBA ORB, and it is free.

1.3.2 GNU C++
The implementation of the runtime repository is done in GNU C++. GNU C++ is supported on a lot of platforms and architectures, and makes the repository less platform dependent.

1.3.3 Linux
Linux was chosen as the development platform. Linux has great support for a lot of programming tools. It is free of charge, but resembles some of the potential target systems for the repository well.

1.3.4 JavaCC
The parser for CQML was implemented with JavaCC. A simple to use, yet powerful compiler generator, for well formed grammars. In addition to being easy to use, it will be a good test on how well formed the grammar of CQML is, and it is free too.

1.3.5 Program and error
The method used for implementing the repository and parser is something you could call program and error. Some people choose to call it extreme programming (XP), but with XP there is extensive use of unit tests, which we drop.

The basic idea is that we switch between designing and implementing pretty rapidly. We divide the problem into smaller parts, make a rough design of a small part, then implement it, and test it. With the experience we gain from this iteration we can go back and improve the design.
1.4 Progression

This work might seem a little dated, especially the references are mostly from pre 2000. This is because this work was started in 1999, and the majority of the work was done in the period from January 1999 to August 2000.

At this point in time and space, what was left to do was to write a few remaining chapters in this report, mainly the implementation, evaluation and conclusion chapters. In the period from August 2000 to February 2007 little work was done to complete the report. The remaining chapters were written during February to May 2007.

Without going into great details about why this is so, a few excuses will be listed. In August 2000, I was enlisted in the army, thinking that it should be possible to complete the report there. In summer 2001 I was employed as a system developer, and in the summer of 2005 I started my own company. This left me with the option of completing the work in my spare time, but with way too much to do and way too many interests, the competition got too tough.

1.5 Acknowledgements

A big thanks to Dr. Jan-Oyvind Aagedal, for guiding me in the right direction, when necessary, and giving very good feedback on my work and writing. He should also be rewarded for his patience. He also provided me with an interesting and rewarding subject for my siding thesis.

Dr. Arne-Jørgen Berre should also receive my thanks, for being my supervisor at the Department of Informatics, and having to cope with me for eight years.

A thanks to my dearest Inger-Marit Østby for being understanding in a period of too much work. She also did some proof reading.

I must also thank the Department of Informatics, and the people at the student office for being very helpful.

1.6 Structure

We will now describe the structure of this document.

In chapter 2 we take a brief look at the theory related to distributed systems and quality of service. This chapter introduces the concepts and terminology used in the rest of this document.

In chapter 3 we list up some requirements that should be placed on a QoS modelling language and the runtime representation of such a language.

Chapter 4 describes some of the related work in this field. We look at the Quality Modelling Language [Frolund 1998a], which has similarities to CQML when it comes to modelling QoS. It also has a QoS Runtime
Representation. We look at the Quality Objects framework [Zincky 1997] which has done some work on adapting to variations of QoS. We then look at the ERDoS framework [Chatterjee 1998], that is a framework for managing resources, and modelling complete real time systems so that they will behave in a more deterministic way.

Chapter 5 describes the basics of CQML. For a more complete description, have a look at [Aagedal 1999]

We then jump to chapter 6 where the design of the runtime system, and CQML parser is described. In section 6.1 we describe an example of a distributed system, that could be a target for QoS management. In section 6.2 we describe the design of the runtime representation of the CQML concepts and the runtime repository. Section 6.3 describes the objects for keeping track of measured QoS, and the base for automatically generated code from the parser. Finally, the design of the CQML parser is described in section 6.4.

In chapter 7 some issues and experiences from the implementation of the runtime system and the parser are discussed. There is also a working example of code generated from the parser that can be used to populate the repository.

Chapter 8 is an evaluation on the runtime and parser against the requirements in chapter 3.

We sum it all up and make our conclusions in chapter 9. In this chapter we also make some suggestions for future work.
Chapter 2

Theory

2.1 Distributed Objects

2.1.1 Distributed systems

A distributed system is defined in “Distributed Systems, Concepts and Design” [Coulouris 1994] as,

a collection of autonomous computers linked by a network, with
software designed to produce an integrated computing facility.

An autonomous computer is a computer that is in control of itself. If the network link goes down, you will still be able to use the computer. A dumb terminal, however, would not work in the case of network failure.

Each computer must have software that enables it to communicate with the other computers on the network. This includes network device drivers, network protocols and object buses.

A distributed system can be homogenous or heterogenous. A homogenous system consists of computers of the same type, running the same system software, and connected by networks of the same type. A system that is not homogenous is heterogenous.

[Coulouris 1994] has identified six key characteristics for distributed systems. They are resource sharing, openness, concurrency, scalability, fault tolerance and transparency. They are all important factors in how good a distributed system is.

Resource sharing A resource is any hardware or software that can be made available on a network. Typical resources are printers, files, databases and programs. A resource is controlled by a resource manager. Access to the resource is done through the manager.

Resource sharing was one of the driving forces for developing distributed systems in the first place. Many resources are (or used to be) quite expensive,
so sharing them would allow a cut in system cost. Also, a group of people working on a project could benefit from sharing their data.

**Openness** Openness refers to how easy it is to extend an existing system. Important factors that influence openness are publishing of interfaces and uniform interprocess communication.

If the resource managers in a system publish their interfaces, it’s easy to add new resources. The newly added resource is made available to other applications in the system by publishing its interface, or by providing another implementation for an existing interface. A laser printer could, for instance, implement the same interface as a line printer.

When all the computers in the system use the same interprocess communication mechanisms, new resources can be added, using that one interprocess communication mechanism.

**Concurrency** Concurrency is a result of several users and processes sharing a system, and thus sharing resources. Several users might try to access the same resource at the same time. Many resources, such as printers, can only be used by one user at a time, so the resource manager must synchronize the access to it.

**Scalability** How large a system can grow, without needing to change the existing system. A 10mbit network can, for instance, only serve a limited number of computers without clogging up the network. Another example is the network address space. For instance, the enormous growth in computers connected to the Internet, resulting in a shortage of IP addresses, one of the problems addressed in IPv6.

**Fault tolerance** In a fault tolerant system, applications should not be affected by resources becoming unavailable. Absolute fault tolerance is not possible. A system with two of everything (two file servers, two network links, etc.) would, however, be quite fault tolerant. However, twice the equipment results in twice the cost, or maybe even more.

**Transparency** There are several aspects of the distributed system the application and application programmer should not have to be concerned with. So, the less of these aspects the application is exposed to, the better.

The following transparencies are described in [Coulouris 1994]:

- **Access transparency.** The application can interact with a resource in a uniform manner, regardless of where the resource is located. For instance, interacting with a local resource will be the same as interacting with a remote one.
Location transparency. The application should not need to know where a resource is located to be able to use it. Instead of printing a document to the /engineering/adm/pc-43/printer, you could simply send the document to the printer named blue-tiger.

Concurrency transparency. An application who uses the same resource as another application, should not have to be concerned with what the other application does. Synchronizing updates to a record in a database should for instance not be the responsibility of the applications using the database.

Replication transparency. An application should be able to use a replicated resource just like it would use a resource that is not replicated. When writing a file to a file server, the application should not have to write a file to the backup server as well.

Failure transparency. A failure in the distributed system would best be handled by the system, without the applications ever knowing. If a network link used by an application goes down, the system should find another route to the remote resource, to enable the application to go on with its work, unaware of the network breaking down.

Migration transparency. Not anyone should notice that an information object moves to a different location. If you talk to a Norwegian on his cellular phone, he would still be able to describe snow, even though someone put him in a box and send him to Africa. You certainly would not know where he was, and he would not even have to know himself.

Performance transparency. The system should be able to reschedule the use of resources to improve performance. It could redirect network traffic to a less loaded link, without the application having to know.

Scaling transparency. The system and applications can grow in scale, without upsetting anything else in the system. You can add more resources without changing the rest of the system. If you add a printer to the system, it would still be the good old, and you have a new place to print stuff.

2.1.2 The client-server model

Distributed systems have up until now been dominated by the client-server model. The client-server model basically consists of a client process and a server process. The client sends a request to the server that the server processes, and then sends a reply to the client. A server may serve many clients, and itself be a client to other servers.
In the client-server model, one server typically controls several resources. This server is called a resource manager, and the clients who wish to access the resources has to do so through this manager.

The fact that a distributed system utilizes the client-server model is in itself no guarantee that the system will score high on any of the six characteristics described in 2.1.1.

The most basic client-server system simply consists of mechanisms for interprocess communication. It will allow resource sharing, and resources can be added. The openness of the system depends on the resource manager programmers. Concurrency will have to be dealt with in the individual servers and clients. Scalability and failure tolerance is more dependent on the lower level network protocols. Hardly any of the transparencies are present.

In a heterogenous system, for instance, how the client interact with the resource depends on what platform the resource manager is running on. How integers are represented might differ between the platforms; some use little endian, while others use big endian. So, for the client and server to communicate, the data has to be converted to something they both can understand. This is called marshalling.

Depending on the marshalling algorithms, the system can be more or less scalable. We thus see that the different characteristics are affected by one another.

There is however nothing stopping us from adding mechanisms to a system based on the client-server model, to be able to build good distributed systems. We can add name servers to provide location transparency, and interface repositories to improve openness.

These mechanisms are collectively known as middleware. Middleware is, as the name states, the layer between the networking layers and the application layer.

2.1.3 Introducing objects

Due to the immense popularity of object-oriented programming, it is only natural that objects also invade the distributed systems. It would be desirable for an object to talk to another object somewhere else in the network, without having to degrade itself by using the more traditional client-server communication mechanisms.

We would in other words like an object to be able to invoke a method on another object at another host, just like it would on an object in the same process. What makes this possible is the object bus. This bus is used by objects to send objects to other objects. The objects that travel the bus can be method invocation arguments and return values, or in the case of multimedia applications, video frames and sound data.

This model allows us to represent resources as objects. To add new
resources to the system, we can simply put them in object wrapping, and
launch them on the network. The object can publish its interface, and it is
available to anyone who wants to use it.

The object representing the resource offers a service to other objects in
the system. It is thus called a server, while the objects using the resources
are clients. Method invocation on server objects is in fact the object way of
the client-server model.

2.1.4 RM-ODP

According to ISO [ISO 1995] there is a need for a co-ordinating framework
for the standardization of Open Distributed Processing, due to the rapid
growth of distributed processing. Their Reference Model of Open Distributed
Processing (RM-ODP) is such a framework.

RM-ODP uses object-oriented modeling concepts to specify distributed
systems. The communication between objects happens at interfaces. This
separation of interface and object allows objects to be interchangeable, and
several implementations can be given for a single interface. This helps deal
with heterogeneity.

The objects can be arbitrarily big and contain other objects. An object
implementing a file server interface could utilize two file server objects to
provide replication. This way, the replication will be transparent to the
application.

To regulate co-operation between objects, we have the contract. A con-
tract specifies the obligations an object has to its co-operating objects, and
what its expectations of them are.

The description of a distributed system can be rather huge and complex.
It is thus necessary to divide the specification into several viewpoints. The
viewpoints defined in RM-ODP are:

- **Enterprise viewpoint** which places the system in a business context.

- **Information viewpoint** which describes the information to be kept in
  the system, and actions to be performed on this information.

- **Computational viewpoint** which describes the system in term of objects
  interacting at interfaces.

- **Engineering viewpoint** which deals with the mechanisms needed to sup-
  port system distribution.

- **Technology viewpoint** which describes the technology used to construct
  the system.

In the computational viewpoint there are three types of interfaces, op-
erational, stream and signal. An operational interface defines the methods
of the interface, a stream interface defines a set of flows supported at the interface, and a signal interface defines the signals supported at the interface. The operations, flows and signals can either be outgoing or incoming. For operations, the interface can be server or client, for streams, producer or consumer, and for signals, initiating or responding. A flow is an abstraction of continuous sequences of data, and a signal is an atomic interaction.

An operation or flow can in fact be explained in terms of signals. An interrogation is made up of invocation emission, invocation receipt, termination emission and termination receipt. Each of these events can be a source for a signal.

An interface has also behavior and an environmental contract. The contract specifies what quality of service (QoS) is expected from an object. It also describes what QoS the object expects of the environment. If the environment fulfills its obligations to the object, the object should also fulfill its obligations to the other objects.

For two objects to be able to communicate with one another, there must exist a binding between them. In the case of an operational interface, the binding can be implicit, it will be established when the client invokes an operation at the server. For stream and signal interfaces, the binding must be done explicitly.

An explicit binding can be primitive or compound. The primitive binding action binds two interfaces of the same type. A compound binding allows several interfaces of the same or different type to be bound. The compound binding is accomplished by the means of a binding object. The binding object supports a set of interfaces, and uses primitive binding to bind to the different interfaces. The binding object may also have interfaces for the management and monitoring of the binding.

Objects and bindings in the computational viewpoint are mapped to basic engineering objects and channels in the engineering viewpoint. The channel consists of stubs, binders and protocol objects. The stub is responsible for marshalling of parameters and anything specific to the information contents and representation. The binder is concerned with the management of the end-to-end connection. It handles connection establishment, communication failures and remote object failures. The protocol objects provide communication at some level of QoS.

### 2.1.5 CORBA

The Common Object Request Broker Architecture (CORBA) is an object-oriented middleware architecture. CORBA consists of an object bus, the Object Request Broker (ORB). The ORB facilitates communication between objects that live on the object bus. Each object on the bus has an unique object reference. Other objects who wish to communicate with the object must know this object reference. When an object uses an object reference
to invoke a method on another object, the ORB makes an implicit binding
between the two objects.

On the client side, there are stubs that marshall arguments into a com-
mon format, and forwards invocations to the object through the ORB. On the
server side, there are skeletons that are extended by the server implement-
ation. The skeletons do the unmarshalling of arguments and marshalling of
return values.

CORBA specifies additional services that are built on top of the ORB.
These services are the Common Object Services, or CORBA Services. The
CORBA Services are a set of normal CORBA objects that extend the ORB.
They are for instance naming, trading and concurrency control services,
adding location, failure and concurrency transparencies, etc. Event and
property are other services that add to the ORB functionality.

CORBA separates interfaces and implementations. An interface can be
implemented in any language, on any platform. Interfaces are defined with
the Interface Definition Language (IDL). The IDL is language and platform
independent. An IDL compiler generates stubs and skeletons for a target
implementation language, that the application or service programmer can
use. This deals with heterogeneity in the distributed system.

2.1.6 Multimedia / Shortcomings

Faster computers and networks are making distributed multimedia applica-
tions more and more viable.

Multimedia applications would typically want to send and receive streaming
data, for example, a camera sending a stream of video frames. CORBA
only supports operational interfaces, and not stream interfaces as defined in
RM-ODP. All interfaces in CORBA are incoming, or server interfaces. There
are no way to specify outgoing interfaces.

There are no signal interfaces in CORBA. Signals are nice for reporting
events to monitoring objects, synchronizing flows in a stream, etc.

CORBA lacks support for explicit bindings. If CORBA had support for
outgoing interfaces, it would be no problem to bind two interfaces. The
two objects implementing the interfaces will however need to use the same
binding establishment procedure, so it would be preferable to have a standard
CORBA binding establishment procedure.

Another aspect of distributed multimedia applications is that the QoS
becomes much more apparent. In a traditional client-server application, a
database lookup could really try your patience. In a multimedia application
you could experience getting a sequence of pictures instead of motion video.
In the end, you would get your data from the database lookup, but the
multimedia application is close to being useless.

Applications would benefit from server objects being able to describe the
QoS they offer. A trader could find the object best suited for the applications
needs based on the QoS the application requires and the QoS the object offers. There are, however, no way to specify QoS in CORBA, except ad hoc methods. There are also not support for contracts between objects.

To summarize, CORBA lacks:

- stream and signal interfaces,
- outgoing interfaces,
- explicit binding mechanisms,
- QoS specification,
- contracts between objects.

In order to use CORBA in a multimedia setting, these shortcomings have to be dealt with.

### 2.2 Quality of Service

There are numerous definitions of Quality of Service (QoS). The meaning of the concept should not be very hard to grasp, though, it simply is the quality of a service. A service we all should be familiar with is the city bus. If we are satisfied with the bus ride, we can say that the bus has good quality of service.

The way we conceive QoS is subjective. One might find the bus ride quite nice, whereas another finds it horrible. We need to quantify QoS, so that it can be measured, and mean the same thing to different people. There can be several aspects of a bus ride that affect the QoS of the ride, for instance how delayed the bus is, there is a seat for every passenger, etc.

QoS will still be conceived differently by different people, but now we can specify the QoS the bus offers, and the QoS we expect from the bus. The bus company might guarantee that the bus should be no more than one minute late. This would be the obligations of the bus to its environment, the people in the city.

The bus might place requirements of its own on the environment in order to meet its obligations. It might require that the roads on the route is open. The obligations of the bus to the environment and the requirements on the environment is an environment contract as defined in RM-ODP [ISO 1995].

It might just happen that this contract is broken. If the environment or object does not fulfill their obligations, there should be specified what actions to be taken. The bus company could guarantee that the bus is no more than five minutes late, and if it is, the passengers will get their money back.
A city bus is simply an object that offers a service to other objects, people, just like an object living in a computer offers services to other objects in the system. The real life QoS can, thus, easily be transferred to the object-oriented computer world.

We need to identify properties that quantify the quality of a service that an object offers. QoS properties for a video camera object could for instance be framerate and colordepth. We then use relational operators to specify the QoS, like \( \text{framerate} > 20 \) and \( \text{colordepth} = 8 \).

An object that wants to use the services of another object must specify the QoS it expects from the service. If the QoS offered by the server object satisfies the client, they can bind and start communicating.

The actual QoS that is delivered can change over time, and deviate from the offered and agreed QoS. We should monitor the binding to find the actual QoS delivered, and if the object does not meet its obligations, action should be taken.

Specification of QoS is subject to ongoing research, and also the focus of this report.

2.3 Object Constraint Language

The Object Constraint Language (OCL) is described in “Object Constraint Language Specification” [OMG 1997]. It is a formal language to express side effect-free constraints.

OCL is used to specify constraints on a system that can not be specified in an interface or object specification, or in a graphical model. A constraint could be that an employee’s age should be higher than 18.

Using OCL we can specify invariants on object and pre- and postconditions for operations. An invariant on an employee object would then be \( \text{age} > 18 \).

OCL is typed. It has the usual built in types, Boolean, Integer, Real and String. There are also collection types, Collection, Set, Sequence and Bag. To be able to reason about collections, there are a number of operations on collections, like selection and iteration operations.

OCL is side effect-free, meaning that an OCL expression will not change the state of an object, nor the global system state.
Chapter 3

Requirements

3.1 QoS modeling language requirements

The focus of this work is on creating runtime support for a QoS modeling language. Developing a QoS modeling language is however not the focus of this work. The requirements in this section are thus not meant to be met by the result of this work. They should rather be thought of as qualities we would like the QoS modeling language to have, from the runtime view.

Those readers interested in the requirements of a QoS modeling language from a system development point of view should have a look at [Aagedal 2001] section 4.3.

1. The language must have constructs to define quantifiable properties that characterize the QoS. This allows the designer to create a vocabulary that can be used to describe QoS.

2. It should be possible to describe the meaning or semantics of such a QoS characteristic. If we want a word to mean the same thing to different people, we have to describe its meaning, precisely. A formal definition of the semantics would ideally allow the runtime system to use different QoS characteristics, that have the same meaning, interchangeably.

3. There must be a way to constrain the values of the QoS characteristics. There is little use talking about measurable QoS if there is nothing to compare the measurements against. The language must thus have constructs to make a statement about the QoS.

4. The language must be typed, so that the runtime system can decide what values of the QoS characteristics give the best QoS. The system should be able to tell whether a statement conforms to another statement.
5. There must be constructs for associating a given QoS with an object, or a binding between objects. It should be possible to specify what QoS an object offers, and what QoS it requires. It should also be possible to have a contract between two interacting objects, specifying the obligation of each object.

6. The QoS modeling language should be a separate language, and not an integrated part of some interface or object definition language. The extensions needed in the interface or object definition language should be kept to a minimum. This to ease the introduction of the QoS modeling language into an existing development environment.

7. As all things of our time, the QoS modeling language should be Object Oriented.

8. It should be possible to combine simple QoS specifications to construct more complex QoS specifications. This would allow us to specify complex QoS, with a limited set of simple QoS specifications. It should be easier to define the semantics of a QoS specification as a function of the semantics of simpler QoS specifications, than to write a huge, complex formal definition from the ground up. This adds to reusability and reduces the possibilities of errors.

9. It should be possible to refine a general QoS specification to make a more special QoS specification. This would for instance allow us to specify the requirements that are common for several objects in one specification. If an object has further requirements, we could refine the common specification to include these additional requirements. This adds to reusability and eases the question of conformity.

10. It is desirable to allow grouping of QoS specifications that are somehow related. Grouping for instance characteristics that deal with performance or security. In addition to categorizing QoS specifications, this also helps solve name conflicts, for instance between similar QoS specifications from different developers.

11. The language should have a well defined grammar. It should be possible to parse in one single sweep, without having to look further ahead than to the next token. This simplifies the implementation of the parser significantly.

12. It should be possible to specify how the application should adapt to changes in QoS. What action should be taken if the agreed on QoS no longer can be provided.
13. It should be a way to specify which constraints are the most important to fulfill. Some constraints might not be as important as others. We might for instance want to degrade video before audio.

3.2 What we require of the runtime system

1. The runtime system should support the same concepts as the modeling language. If the runtime system uses another set of concepts to specify QoS, we would need a map that maps modeling language concepts into runtime system concepts. This would indeed add to the complexity of the QoS framework, be a source of errors, and it would be harder for the system designer to see that the implemented system in fact is what he designed.

2. The runtime system should impose as little overhead as possible. The benefit we get from managing QoS should be greater than the cost of the resources needed to manage QoS. If the QoS manager decreases the overall QoS of the system to a point where it would no longer be of any use to its users, the system would probably be better of without the QoS manager. QoS management is after all not the core functionality of a system.

3. The design of the system should be platform independent. If the design does not depend on any platform specific features we can reuse the design if we move to another platform. The QoS runtime system would thus cover a larger range of systems. A developer would have the same general interfaces to program against.

4. The implementation of the system should be platform independent. A distributed system is likely to be composed of different technologies, and would have to communicate across these different platforms.

5. Communication between objects specified in the interface or object definition language and the QoS runtime system should be kept to a minimum, and only through well defined interfaces. If a system uses features that are specific to a particular implementation of the QoS runtime system, it would be harder to use another QoS runtime system implementation.

6. The system should allow for both design time and runtime specification of QoS. QoS characteristics and general QoS statements is likely to be specified at design time, whereas each user should be able to specify the QoS he wants at runtime.
7. The system should have support for anonymous QoS specifications, to allow the load of specifications from the users and system, not known at design time.

8. The system should support parameterized QoS specifications. Two objects that have different requirements to the same QoS characteristics could have QoS statements of the same type. This could be used for optimization purposes in the QoS runtime system. An object that changes its requirements or offers concerning QoS often, could change an attribute in the QoS statement instead of creating a new QoS statement.

9. The runtime system should have support for measured QoS. To be able to control that the agreed on QoS is delivered, the real QoS must be compared to the specified QoS. This is only possible if the system knows the real, or measured QoS. This requirement follows from the above requirements.
Chapter 4

Related work

4.1 QoS Modeling Language

The QoS Modeling Language (QML), [Frølund 1998a], is a language for specifying general purpose QoS. QML uses three abstractions to specify QoS, contract type, contract and profile.

A contract type is used to represent a category of QoS. The contract type defines dimensions that are used to specify the QoS for that category. A dimension has a domain of values, that might be ordered. The dimension can be increasing or decreasing, specifying that bigger values are better, or vice versa. The dimension also has a unit. The domain can be a numeric, enumerated or set domain.

A contract is an instance of a contract type. A contract represents a QoS specification. The values for a dimension is constrained, using plain greater than or less than constraints, or using statistical properties, for instance mean, variance and percentile. The profile is used to bind contracts to interface entities.

An example, partly taken from [Frølund 1998b]:

interface RateServiceI {
    Rates latest(in Currency c1, in Currency c2);
};

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

systemPerformance = Performance contract {
    delay {
        percentile 80 < 20 msec;
    };
}
percentile 100 < 40 msec;
mean < 15 msec;
throughput > 100;

rateServerProfile for RateServiceI = profile {
  require systemPerformance;
};

The interface is defined in some interface definition language, separate from QML. We then define a contract type, Performance, with dimensions delay and throughput. A contract is then defined that constrains the delay and throughput. rateServerProfile binds the contract to the interface. This profile specifies a default contract that constrains all entities of the interface.

As we have seen, QML separates the interface specification from the QoS specification. QML can thus be used in conjunction with several different interface definition languages. The separation allows us to specify several contracts for a given interface. This means that different implementations of the interface can have different contracts bound to them. A client can then choose the implementation with the contract that fits it best.

A server that implements an interface with an associated contract, specified in a profile, offers the QoS constrained by the contract. A client can use a profile to specify the QoS it requires of the server object.

If the offered QoS is better than the required QoS, we say that the profile supported by the server conforms to the profile required by the client. Profile conformance is based on contract conformance, which again is based on constraint conformance.

Contracts and profiles can be refined. A refinement is used to make a contract or profile stronger. The refined contract will conform to the base contract. We can for instance use refinement to override the default contract for an operation or operation argument.

4.1.1 QoS Runtime Representation

QoS Runtime Representation (QRR) is the QoS fabric of QML. QRR specifies the runtime representation of QoS specifications, and library functions.

The requirements of QRR set forth in [Frølund 1998b] are:

1. QRR should support the same fundamental concepts as QML.
2. It should be possible to dynamically create new QRR specifications.
3. It should be possible to check consistency of created specifications against the static semantic rules of QML/QRR.
4. It should be possible to manipulate existing QRR specifications.

5. The overhead imposed by QRR should be minimal.

6. Provide a minimal set of mechanisms to specify QoS.

Contract types, contracts and profiles are represented by IDL structs.
There are two alternative representations for contract types and contracts, static and generic. In the static representation, a type (struct) is defined for each contract type. Only contracts of the same contract type are of the same type. In the generic way, all contracts are of the same type.

There are library functions to create contracts and profiles, manipulate them, and checking them for consistency, etc.

QML generates structs for representing contracts and profiles, and functions to manipulate them.

The static representation is faster for manipulating, and especially conformance checking. The generic representation allows for dynamically creating contracts that are not known at compile time.

4.2 Quality Objects

The Quality Objects (QuO) framework [Zinky 1997] helps distributed application developers build adaptive QoS aware systems. QuO extends the CORBA Interface Definition Language (IDL) with a suite of languages, collectively known as QoS Description Language (QDL). The object designer can specify the interface to an object in IDL, hiding implementation details. In QDL, he can open up the implementation, exposing important implementation details.

QDL can be used to set up a contract between the client and a remote object, in terms of expected client usage and promised QoS by the remote object. A QoS contract is split up into QoS regions, allowing the developers to specify object and client behavior in each region, and what to do in case of region transitions. The region is defined by predicates on system properties. System properties are encapsulated in first class objects. Network level QoS properties, for instance delay and throughput, are translated to object level QoS properties, such as method invocations per second. QDL contracts, behavior policies and system property objects can easily be reused by other applications.

QuO tries to reduce the variance of system properties by using layers of delegate objects, in the client address space. Each delegate uses the knowledge it possesses about the layer to improve the system properties seen by the layer above it. If throughput is low, the object delegate might choose to compress the data before sending to remote object. This is all done transient to the client.
4.2.1 Connection

Connections are first class objects in QuO. The connection forms a boundary where expected usage patterns and QoS requirements between client and server object can be agreed upon as contracts.

The responsibility of end-to-end QoS is placed on the server object. This way, the server object designer can specify what to do if the connection to the remote server object should go down, instead of placing that burden on the client designer, who has less knowledge about the server object.

The server object is represented in client address space by an object delegate, an object having the same interface as the remote object. At connect time, the client will create this local delegate, which will connect to the remote object. The client can thus communicate with the remote object, as if it was a local object.

4.2.2 Contract

The contract is made up of negotiated regions, which themselves contain reality regions. A negotiated region is a named region defined by client usage and object QoS expectations. A reality region is a named region defined by measured client usage and object QoS.

Whenever there is a transition from a reality region to another, a handler can be called, giving the client and object the ability to adapt to changing system conditions. A transition from a negotiated region to another signals that the agreed upon QoS cannot be achieved, and more dramatic actions are called for, like contract renegotiation.

The regions are defined by predicates over system properties. The regions are ordered by a precedence relation.

4.2.3 System properties

QuO represents system properties as first class objects, called system condition objects. The interfaces to the system condition objects are specified in IDL. Most system condition objects are simple attributes, with read and write methods.

The system condition objects function as interfaces between the different delegate layers. The system properties of a layer will be translated to properties more suitable to the layer above, and the system condition objects of the layer above will be updated accordingly.

System condition objects can also be more complex, for instance monitoring the status of some peripheral, alerting a delegate object if some threshold is crossed.
4.2.4 Adapting

An application can deal with varying system properties in several ways. It might accept that the method returns later than expected, use an implementation of the method that does a little less work, or utilize other mechanisms, such as compression.

When the expected QoS is unlikely to be met, the client and server object can be notified of this through a reality region callback. They can also be notified by the negotiated region that the QoS is unlikely to be met. They can adapt by calling different functions internal to the different regions, or the object designer can specify alternate paths in the connection, using the Structure Description Language (SDL).

When a delegate no longer can maintain the QoS required by the current negotiated region, it notifies all layers above of a reality region change. All parties try to adapt to the change in QoS, by changing policies, using alternate implementations, or maybe other servers. If the parties cannot adapt, the delegate will notify the layers of a negotiated region change.

4.2.5 Delegates

The delegate objects will be generated by a code generator, as specified in the Contract Description Language (CDL). We have an ORB delegate, object delegate and client delegate. The ORB delegate is specific to the ORB, and has information on how to best make adaptions in that specific ORB. The object delegate is specific for the object, and knows about mechanisms in the object that can be used to adapt to changing QoS, for instance compression. The client delegate is not required, but can be used to choose among different remote objects, having different resource requirements and behavior.

4.2.6 Aspect languages

Instead of having one big language for specifying everything QoS related, QuO defines one language for each aspect of the application. There is one language for describing contracts, Contract Description Language (CDL), one for describing the internal structure of an object implementation, Structure Description Language (SDL), and one for describing resources, Resource Description Language (RDL). A code generator weaves the different parts together, generating code for the different objects in the QuO runtime system, contract objects, QoS region objects, system condition objects, etc.

The contract, defined in CDL, will define all the QoS regions, with expected usage and required QoS. There will be hooks for region transition callbacks, and parameters to customize the regions. The client will create a contract, pass handlers for client callback and client expectations.

SDL is used to specify how the object will behave for a particular method call and return, in a particular QoS region of the connection. You can for
instance specify what object implementation to use, one that has replication mechanisms versus one that has not.

A more thorough description of the aspect languages can be found in [Loyall 1998].

4.3 ERDoS

In this section we describe the End-to-End Resource Management of Distributed Systems (ERDoS) QoS Architecture [Chatterjee 1998]. The modeling of a distributed system is divided into three perspectives: application, resource and system.

An application is interested in achieving the highest possible level of QoS, the effect on other applications being irrelevant. This perspective is modeled by the Logical Application Stream Model (LASM), that captures the structure of an application in a system independent manner, and the Application Invocation Model (AIM), that contains application and user QoS preferences for an instance of the application. The AIM contains a Benefit Function (BF) that models the benefit a user receives for different values of received QoS.

The resource perspective is internal to the management of a resource. It is not concerned with other resources or applications running on other resources. The Resource Model captures information about individual resources. It has a resource specific set of attributes, and a resource independent set of attributes, to allow the description of resources in a uniform manner.

The system consists of resources and applications. It is the responsibility of the system to provide end-to-end QoS support to the applications, while utilizing the resources efficiently. The system must implement policies for resolving conflicts between applications and resources. For instance what applications and what QoS to degrade when there is insufficient resources. The System Model (SM) captures the resources the system manage, and the topological and administrative structure of the system. This includes the policies and objectives of each of these administrative domains.

4.3.1 QoS Taxonomy

In the ERDoS architecture, QoS parameters are grouped into metrics and policies. The metrics measure quantifiable QoS attributes, and are further divided into performance metrics, security levels and relative importance. The policies describe the system behavior specifications.

The performance metrics are divided into timeliness, precision and accuracy metrics. Timeliness is expressed in units of time, and is used to express timing requirements, such as delay, start time, deadline, etc. The precision
parameters specify volume related quantities. This can be number of digits in a floating point number, or the number of bits used to represent the number. Accuracy measures the error introduced into the data by services. Number of valid digits in a floating point number is an example. There are cases where QoS can be specified as a combination of these metrics, for instance throughput, defined as precision over time.

There are two security metrics, level of confidentiality and level of integrity. The level of confidentiality ensures that the data does not fall into the wrong hands, whereas the level of integrity ensures that the data is not tampered with by unauthorized personal or processes.

The relative importance tells the system which application to degrade first, in case of resource shortage.

The availability policy specifies the level of availability the system provides, and the application requires. The system can be best effort or guaranteed. In case of a guaranteed service, we can define levels of guarantee.

The management policies tell what actions to take in case of unforeseen events. For instance, whether to renegotiate or abort when the QoS is degraded.

4.3.2 Logical Application Stream Model

The LASM captures the structure and relevant QoS parameters of an application, in a system independent manner. There is a single layered model, and a recursive model.

An application can be seen as a set of tasks executing in a specific order on a set of resources. An atomic task is denoted a Unit of Work (UoW). A UoW will transform the data of the application, and from the resource management perspective, the QoS of the application.

An application is modeled by a directed graph. In the single layered model, the nodes are Logical Units of Work (LUoW), connected by directed edges, denoted Logical Edges (LE), that dictate the order in which the LUoWs execute.

In the recursive model, the nodes are Logical Services (LS) connected by LEs. An LS is realized by a Logical Realization of Service (LRoS). The LRoS consists of either a single LUoW or a set of child LSs.

Application QoS relationships are captured by the Logical Constraints (LC) set. QoS attributes that must be assigned dynamically, must be declared in the Logical Parameter (LP) set. The LC is used to assign values to LUoW attributes.

4.3.3 Resource Model

A resource exports a set of Physical Units of Work (PUoW). A PUoW specifies a LUoW that the resource can execute. A PUoW is similar to a LUoW,
but in addition it specifies the load that executing this PUoW will have on the resource.

The resource independent attributes of a resource are resource scheduling model, resource utilization, resource load and resource reliability.

The system uses these attributes to find the best resources for an application.

4.3.4 System Model

The system model describes the structure of the system using a graph. Resources are presented by nodes, and the edges represent connections between resources.

The system is hierarchically decomposed, with the resources at the bottom layer, encapsulated by subsystems. At the top level is the system, which is composed of subsystems. A subsystem can either contain more subsystems or resources. The top level system is responsible for scheduling applications and providing QoS guarantees.

This hierarchical model describes the organizational topology of the system, and each subsystem has its own manager, implementing the policies for the subsystem.

4.4 QMF

4.5 Summary

QML is a simple language that enables us to quickly describe the QoS aspects of a system.

QuO and ERDoS are more complex modelling tools, that enables us to make a more complete model of the system and QoS. Thus, making it more complex, and harder to learn. The resulting systems might be more deterministic.
Chapter 5

CQML

The QoS runtime framework described in this report supports the same concepts as the Component Quality Modeling Language (CQML) [Aagedal 1999]. This is an early version of CQML, and the final version can be found in [Aagedal 2001]. Because the final version was not available when this work started, this work is based on the version in [Aagedal 1999].

The fundamental concepts used to describe QoS are QoS characteristic, QoS category and QoS statement. A QoS characteristic describes a property that characterizes the QoS, for instance end-to-end delay, or throughput. QoS categories are used to group QoS characteristics. Delay and throughput could for instance be grouped into a timeliness category. QoS statements are used to constrain QoS characteristics, for instance stating that delay should be less than 10 ms.

An object can specify what QoS it offers and requires through provides and requires clauses in the definition of the object. CQML does not specify any object definition language, but simply adds the provides and requires clauses to some existing language, for instance TINA ODL [TINA-C 1996]. An object will provide the QoS specified in the provides clause, as long as it receives the QoS specified in its requires clause.

A QoS contract is used to describe what QoS the co-operating objects provide to each other. We can specify that one object provides the QoS stated in a QoS statement to another object.

In the following section, we will describe the various concepts more thoroughly.

5.1 QoS characteristic

A QoS characteristic is given a unique name. The definition of a QoS characteristic is a type definition. This enables us to type check constraints.
5.1.1 Domain
A domain is defined for the QoS characteristic. There are three kinds of
domains, numeric, enumerated and set. The domain can be ordered, partially
or totally. And we can specify whether higher values are better than lower,
increasing, or vice versa, decreasing. The QoS characteristic might also have
a unit associated with it.

//A simple example

class qos_characteristic delay {
  domain: decreasing numeric ms;
};

The example defines a QoS characteristic named delay. It is defined over
the numeric domain, lower values are better, and the unit is milliseconds.

Thus far, a QoS characteristic is pretty much the same as a dimension in
the contract type definition in QoS Modeling Language, described in 4.1. I
quote, “A contract type defines the dimensions that can be used to charac-
terize a particular QoS category.”, [Frølund 1998b].

5.1.2 Statistical aspects
Another similarity is that you can specify QoS in terms of the QoS charac-
teristic itself, i.e. delay < 10, or in terms of statistical aspects. You can
specify what statistical aspects should be available. If you do not specify
any, all aspects are available. The statistical aspects are, maximum, min-
imum, range, mean, variance, standard deviation, percentile, moment and
frequency. The three last ones take parameters.

//Simple example with statistical aspects

class qos_characteristic statistical_delay {
  domain: decreasing numeric ms;
    mean;
      percentile 90;
};

In the above example, we define that delay can be specified in terms of
mean delay and the 90th percentile, that is, the best 90%.
5.1.3 Specialization

The statistical\_delay QoS characteristic differs from delay only in that it has specified two statistical aspects. We can say that statistical\_delay is a specialization of delay.

```plaintext
//Specialized characteristic

qos_characteristic statistical\_delay : delay {
    mean;
    percentile 90;
};
```

The specialization is now explicit, which can be useful later, in type checking and conformance tests. We say that statistical\_delay refines delay.

5.1.4 Semantics

In CQML we can also define the semantics of a QoS characteristic. The semantics are defined by OCL expressions.

We can add invariants to QoS characteristics, to constrain the values in the domain, or specify relationships between statistical aspects. Delay should for instance be non-negative, \textit{minimum} >= 0.

To describe the semantics of QoS characteristics that deal with timeliness, we associate events with the emission and reception of data. In the computational viewpoint of RM-ODP, [ISO 1995], an interface is either an operational, stream or signal interface. The operations, flows and signals are either incoming or outgoing. We associate an event sequence with the emission and reception of data for each operation, flow and signal, one SE and one SR sequence. SE is for emission, and SR is for reception.

OCL has a built in container type, Sequence. EventSequence is defined as a specialized Sequence, holding Events. SE and SR are of type EventSequence. When an operation, op1, is invoked, an event is added to the op1.SR sequence. When op1 terminates and returns the return values, an event is added to the op1.SE sequence. A function, time(e:Event), returns the time of the event.

To be able to reason over interface entities in a QoS characteristic definition, we need parameters. Interface entities can then be passed to the characteristic as arguments.

```plaintext
//I feel an example is needed
```
The equality relation on events is defined to be true if the two events correspond to the same data. The expression iterates through all the events in seq2, finds the corresponding event in seq1, and returns the time difference of the two events with the biggest difference. The delay is thus defined as this maximum time difference between corresponding events in the two EventSequences.

It is possible to define utility operations using OCL. This way, complex expressions can be broken down into smaller components.
maxTimeDifference is an operation that returns the time difference between
the two corresponding events in seq1 and seq2 with the biggest difference.
The operation is then used to define the semantics of the QoS characteristic
delay.

5.2 QoS category

A QoS category is used to group QoS characteristics. Delay and start time is
for instance two characteristics that can be grouped in a timeliness category.

In 5.1.1 we related the QoS characteristic with a dimension in a contract
type in QML. It should be obvious that a QoS category corresponds to the
contract type.

5.3 QoS statement

Also the QoS statement has a counterpart in QML, the contract. A QoS
statement has a name, and constrains one or more QoS characteristic.

//low delay

qos fast(seq1 : EventSequence, seq2 : EventSequence) {
  delay(seq1, seq2) < 5;
};

//or in terms of statistical aspects
qos not_so_fast {
    statistical_delay.mean < 20;
    statistical_delay.percentile 90 < 25;
};

The fast QoS statement states that delay should be less than 5. It takes parameters, that it passes to the delay characteristic. The statements take parameters, just like QoS characteristics. In the not_so_fast statement, the average delay should be less than 20, and for the best 90 percent, delay should be less than 25.

5.4 QoS offers and requirements

CQML should be used together with a computational modeling language, for instance TINA ODL, [TINA-C 1996]. We define interfaces and object templates in the computational language (ODL). The object template definition is extended to include provides and requires clauses.

A provides clause specifies what QoS the object offers. It specifies what QoS statements the object should satisfy. Several QoS statements can be specified with and/or expressions.

To fulfill its QoS obligations, set forth in the provides clause, the object requires that some QoS is provided to it from the environment. What QoS the object requires is specified in the requires clause. The requires clause is similar to the provides clause.

As an example, we define an interface to some military application, an object template and specify the required QoS. The interface and object definitions are in ODL, enhanced with requires and provides clauses.

interface RangeCalculator {
    double rangeToTarget(Target t);
};

object RangeOmatic2000 {
    supports RangeCalculator;
    provides fast(RangeCalculator.rangeToTarget.SR,
                 RangeCalculator.rangeToTarget.SE);
};

The RangeCalculator interface has an operation to calculate the distance to a target. The RangeOmatic2000 implements this interface, and promises
that the time from the invocation of the function, to the return of an answer should be less than 5 milliseconds.

### 5.5 QoS contract

We could have another object, for instance a weapons system, that uses the RangeOmatic, that have requirements on the QoS of the rangeToTarget operation. If the QoS offered by the RangeOmatic2000 offers conforms to the QoS the weapon system requires, a contract can be made between the two.

With CQML we can specify this contract already at design time. That is, we can specify that the RangeOmatic2000 must always offer this QoS to the weapon system, because its kind of critical.

```qos_contract
  calculations {
    RangeOmatic2000 provides
      thousand_calculations_per_second to
      Thomas_the_tank;
  }
```

In the example, the RangeOmatic2000 promises that it can do thousand calculations per second for Thomas_the_tank. The QoS statement is defined elsewhere.
Chapter 6

Design

The framework can be logically divided into three parts. The QoS repository, where QoS characteristics, statements and temporary QoS objects can be created. QoS utility objects that lives in client or server space, and aids the application with negotiating and monitoring QoS. The CQML parser, which parses CQML files and generates code for use at runtime, and populates the repository. We will describe the design of these components in this chapter.

6.1 Architecture

The architecture of the QoS framework is not the main topic of this work. However, in this section, we will describe an example application that could be using the repository.

The design of this application will be described using concepts from ODP. In this simple example, we have one object that wants to use a service of another object. These two objects communicate through a binding object. This is illustrated in figure 6.1. The binding must reflect the interface of the server to the client.

So far, we have designed a functional model for the application. The

![Figure 6.1: Application example](image)
client does, however, have requirements on the QoS of the server. It could for instance require that a method that performs some calculation returns its response within a given amount of time. The delay of this response is the sum of the time the server uses to do the calculation, the time it takes for the request to reach the server and the time it takes the response to get back to the client.

To give the client what it requires, the binding object must determine the delay in the network transmissions, and it will have to place some constraints on the server object. In the following, we concentrate on the QoS of the interface between the binding and the server.

For the system to be able to manage its QoS, the QoS architecture needs some information objects in the QoS repository, and some runtime QoS management objects, which are explained in the following.

### 6.1.1 QosRepository information objects

The binding object has requirements on the server object, which it will expose through a `require` clause. The server should for instance be required to fulfill the QoS statement, `qos_fast`.

To be able to specify that the binding object requires `qos_fast` we create a `qos_fast` object in the repository, which is a `qos_statement`. The binding object will hold a reference to this `qos_statement` object in the repository.

For a negotiator to be able to match a binding object with a server object, it would have to confirm that the server object will provide the QoS that the binding object requires. If the server for instance provides the `qos_statement` `qos_lightning`, we have to create a `qos_lightning` object in the repository, and have the server object keep a reference to it.

The two `qos_statement` objects in the repository will have to specify the QoS by constraining some QoS characteristic. Let us say that they both place constraints on delay. We create a `qc_delay` object in the repository, that both `qos_statement` objects use.

Now we have all we need in the repository to get the QoS-aware connection started. There are, however, chances that the QoS will not stay the same all through the entire communication session of the two parties involved.

The system is supposed to adapt to changing QoS. The system will thus have to know the QoS of the system at any given time. We have to measure the QoS, which in this case would be the delay. We create a new object in the repository. This object will keep track of the delay that the binding object sees, and it should calculate values for the different statistical aspects of the corresponding `qos_characteristic`. This object is a `qos_real` object. One may think of it as a `qos_statement` on the observed QoS of the system.

The objects in the repository can be seen in figure 6.2, together with some QoS management components that we will see in the next section.
6.1.2 QoS management components

When it comes to runtime management of QoS, there are two issues that must be dealt with. First we must negotiate a contract between the two communicating parties. Second we must monitor the QoS, to see to that the QoS agreed on in the contract is delivered. In case of a broken contract, measures must be taken, for instance renegotiation.

Before the parties can start communicating, they must agree on the QoS. For the communicating objects to be able to negotiate an agreement on QoS, they need a way to speak to each other. We use a Negotiator interface for this purpose. The objects must either implement the Negotiator interface themselves, or have some Negotiator object represent them in the negotiations. The object must expose its Negotiator object through some method. This is elaborated later in this section.

The negotiation can be a complex operation, especially when several objects are involved. The QoS the server can provide may depend on the QoS it receives from other objects. In our simple example, however, we will simply have the Negotiator compare the requires clause of the client object to the provides clause of the server object, to check for conformance.

When the two parties have agreed on the QoS, the binding object is often interested in knowing that it gets the QoS it is promised. To be certain of this, it can create a QoSMonitor object that is used to monitor the QoS. If the QoS the binding object receives is not the same as that promised, the QoSMonitor object can alert the binding object that something is wrong.

The QoSMonitor will register itself as a listener for events from the binding object. Whenever the binding object calls a method on the server object or receives a response from the server object, it will emit a notification event. These events make up the eventsequences as described in CQML.

The QoSMonitor will use the events it receives to calculate values for the different QoS characteristics involved. It will then send the value for the qos_characteristic to some qos_real object in the repository.

A qos_real object, is a repository object, that represents some observed QoS. qos_real inherits from qos_statement, and we can thus check that the observed QoS conforms to a qos_statement. You can say that qos_real is a statement on the observed QoS. For instance, qc_delay equals 4ms.

In the example, the QoSMonitor will calculate a value for qc_delay, and send it to the qos_real object, measured, in the repository.

The complete model for this example can be seen in figure 6.2.

To ensure that all things are well, the QoSMonitor will check that the measured QoS conforms to the QoS agreed on in the contract. If the negotiation led to a contract stating that the server object should provide the qos_fast statement to the binding object, the QoSMonitor will have to verify that the call to measured -> conformsTo ( qos\_fast ); evaluates to true.
Figure 6.2: QoS managed application
If the measured QoS should happen to not conform to the agreed QoS, the QoSMonitor will emit a breach of contract event to all parties interested.

An example of QoS negotiation and monitoring is shown in figure 6.3. In this example, the server does not deliver satisfactory QoS and the binding object is notified.

To make this QoS management work, we need to make the objects in the system QoS aware. In other words, the objects need a way to communicate QoS information. The server and client objects need to speak with each other during negotiation. An object has to speak to the repository and to the QoSMonitor.

For an object to be QoS aware, it must implement the QoSObject inter-
face. It is through this interface you can get a reference to the Negotiator, and register as a listener for events from the object.

6.2 Repository

In this section we describe the design of the QoS repository. The repository is a container for storing QoS specifications. In figure 6.4 the interfaces to the objects in the repository is illustrated.

The repository is used both at design time and run time. At design time, the system designer can use the repository as a QoS specification catalog, using already defined QoS specifications, and inserting new ones, specific to the particular project he is working on.

At run time, the QoS aware objects in the distributed system use the QoS specifications in the repository when they negotiate QoS. The repository also has support for representing the real, measured QoS of a system.

In the following sections, we describe the responsibilities and methods of each object that inhabits the repository.

6.2.1 Simplifications

In the CQML language, you can specify parameters for the qos_characteristic and the qos_statement. This is not supported by this version of the repository.

This simplification makes it impossible to have qos_statements with two or more constraints on the same qos_characteristic. However, keeping the qos_statements simple, with few qos_characteristics makes them better suited for reuse.

Including parameters for qos_statements and qos_characteristics should not be to hard. The parameters are, however, heavily involved in the semantics of the qos_characteristics. Representing the semantics in the repository is a complex and time demanding task, and will have to wait for a future version.

6.2.2 QRObjec

All objects that live in the repository is of this type. This allows us to treat all objects in the repository in a uniform manner, for instance by returning all the objects in the repository as a sequence of QRObjec.

• readonly attribute DefinitionKind defKind

This attribute specifies what type of entity this object represents.
6.2.3 Container

Responsibilities  One of the main tasks of the repository is to contain other objects. We, therefore, need an object that can contain other objects. We also need to create objects that should live in the repository. One possibility is to create the object outside the repository, and then import it, another is to have a factory object in the repository that creates the objects.

We choose the latter, as it does not waste time, creating an object outside the repository that at last is going to live in the repository after all.

The best would be to have a separate interface for factory objects. A container could have a method that returned a factory object, that would be used to create objects in the container. This would be more modular, allowing containers to have specialized factories, for example due to performance considerations.

We choose to combine the container and factory in one interface, thus making it the responsibility of the container to create its own objects. It is simpler, and sufficient to create a functional repository. A factory interface could be added at a later time anyhow.

The task of the container is simply to create and store the objects. We leave for the objects themselves to add content to themselves.

An object that is supposed to contain QoS specification objects must implement the Container interface.

The responsibility of a Container object is to store and manage other objects.

- The container must provide persistent storage for other objects that implement the Contained interface.
- It must have methods for creating objects and removing objects from the container.
- It must have methods to look up objects by name, and return a list of all the objects in the container.

Behaviour

create  The container has five methods for creating objects inside itself. One method for each one of the QoS specification objects, create_qos_characteristic, create_qos_statement, create_qos_category, create_qos_real and create_qos_reference, for the respective QoS objects. These objects are specified later in this section. The methods return a reference to the newly created objects.

All five create methods take an Identifier as argument. The Identifier is simply a string, serving to uniquely identify the object within the container. If there already is an object with the same name in the container, the method throws a NameCrash exception.
If the method is passed a nil reference as the Identifier, it creates an anonymous object. Anonymous objects are useful for specifying QoS that has been negotiated by two parties. This way we avoid name crashes for QoS specifications that are only useful for the parties involved, for the duration of the contract between the communicating parties.

In addition to the identifier, the `create_qos_characteristic` takes a `DomainKind` as argument. The `DomainKind` is an enumeration, defining numeric, enumeration and set domains, whether they are increasing or decreasing, and if they are ordered. The reason for this is that the domain must be known before values can be defined for the domain.

The `create_qos_reference` takes a reference to another `Contained` object as argument. The object created will be a reference to this object. Operating on this reference should have the same result as operating on the object itself.

**find** The container has a method for looking up objects by name, `find_name`. The method takes a `ScopedName` as argument. The `ScopedName` is the unique identifier of the object we want to find. For the lite version of the repository, the `ScopedName` must be the same as the identifier of an object in this container. For later versions of the repository, the `ScopedName` can be a path to an object, relative to the container.

If the named object is not found, a nil reference should be returned, otherwise the reference to the object associated with the name should be returned.

**list** The container has a method that returns a sequence of references to all the objects in the container. This method is useful when the repository is used as a catalog of QoS specifications, for instance as a design tool.

**remove** The container has a method for removing objects from the container. The method takes a reference to the `Contained` object to be removed.

### 6.2.4 Contained

The purpose of the `Contained` interface is to be a common superinterface to all objects that live in a container, so the container can treat the different objects as uniform as possible.

The `Contained` object has one method, `get_container`, that returns a reference to the container the object is living in.

It also has one readonly attribute, `id`, which is the unique identifier for the object within its container.
6.2.5 QoSRepository

The QoSRepository is the main entrance to the repository. For the lite version, its sole purpose is to be the root container of the repository. Later versions can add functionality that is specific to the repository, but not to other containers.

When a client gets a reference to a QoSRepository, it will implicitly get the reference to the root container of the QoSRepository.

6.2.6 qos_category

The purpose of the qos_category is to group related QoS specifications together. The qos_category inherits the Container interface, so it can contain other named Contained objects. It also inherits the Contained interface, so it can itself be contained in another container, or qos_category.

We can thus build a tree of QoS specifications. The qos_category behaves pretty much like a directory in a file system.

Example  We will create a namespace for our company, and in that namespace define two categories, timeliness and security.

```cpp
QoSRepository_var repository =
    // get reference to repository somehow.

qos_category_ptr cat_company;
cat_company = repository -> create_qos_category("qosRUs");

qos_category_ptr cat_time;
cat_time = cat_company -> create_qos_category("timeliness");

qos_category_ptr cat_security;
cat_security = cat_company -> create_qos_category("security");
```

6.2.7 qos_characteristic

Responsibilities  The main responsibility of a qos_characteristic object is to represent a definition of a qos_characteristic, as specified in CQML. The object must further have methods that allows it to be used as intended by CQML.

We saw in a previous section that the creation of a qos_characteristic object was the responsibility of the Container. It is the responsibility of the qos_characteristic itself to provide methods for building its contents. We thus need methods that correspond to the constructs used to define a
qos_characteristic in CQML. For instance for defining refinement, statistical aspects, and so on.

There must be methods that can be used to inspect the qos_characteristic object. These methods are needed by a qos_statement object when used to constrain the values of a qos_characteristic. When a qos_statement should place constraints on a statistical aspect of a qos_characteristic, it must verify that the statistical aspect is defined for the qos_characteristic. It must also verify that the constraining value is in the domain of the qos_characteristic.

The objects must have a method for comparing two values in the domain of the qos_characteristic, to decide which one is better. This method is necessary when confirming whether a qos_statement conforms to another, or not.

At last, the qos_characteristic must have a method for checking that two qos_characteristic objects are the same. This is also needed when checking qos_statement conformance. It would be desirable to have the method check that two objects are semantically equal. This way, we could use two qos_characteristics defined independent of each other, but which have equal semantics, interchangeably.

For the lite version, it is only required that the method checks that the two qos_characteristic references point to the same qos_characteristic object in the repository.

Behaviour

refinement The qos_characteristic has a method for defining that this qos_characteristic is a refinement of another, refines.

domain There are two methods for defining the domain. In the previous section, we described that the kind of the domain was defined as an argument to the create_qos_characteristic method of the Container object.

The two methods used to define the domain are domain_values and domain_order, for set and enumerated domains. We use the domain_values method to define the elements in the domain. The domain_order is used to order the elements relative to each other. The domain_order method takes a list of (low, high) value pairs, which states that the low element is lesser than the high element. This way, we can define a total or a partial ordering of the elements.

The interface also has a method to check if a value is in the domain of the qos_characteristic, value_in_domain.

statistical aspects The interface has two methods for defining the statistical aspects of the qos_characteristic. It has one for simple aspects, and
one for parameterized aspects, \textit{simple\_aspect} and \textit{param\_aspect}, respectively. It further has two methods for checking whether a statistical aspect is defined for the qos\_characteristic. Again for simple and parameterized aspects, \textit{simple\_aspect\_defined} and \textit{param\_aspect\_defined}.

\textbf{object equality} The qos\_characteristic has a method for comparing the qos\_characteristic with another for semantical equality, \textit{has\_equal\_semantics}. It takes a reference to the other qos\_characteristic.

For the lite version it is only required that the qos\_characteristic checks that the reference passed as argument references the qos\_characteristic itself. It is thus possible that the method returns false for two semantically equal qos\_characteristics.

\textbf{compare} The qos\_characteristic has a method for comparing two values in the domain, \textit{compare}. The method takes two values as arguments, strong and weak. It returns 0 if the values are equal, a positive integer if strong is better than weak, and a negative integer otherwise.

\textbf{Examples} To illustrate how a qos\_characteristic object is built, we will now insert the delay and the statistical\_delay characteristics from the examples in sections 5.1.1 and 5.1.2.

```cpp
qos\_characteristic\_ptr qc\_delay;
qc\_delay =
cat\_time \to create\_qos\_characteristic("delay",
       decr\_numeric);

qos\_characteristic\_ptr qc\_stat\_delay;
qc\_stat\_delay =
cat\_time \to create\_qos\_characteristic("statistical\_delay",
       decr\_numeric);

qc\_stat\_delay \to simple\_aspect(ak\_mean);

AspectParam param;
param\.kind = pk\_number;
param\.number = 90;
qc\_stat\_delay \to param\_aspect(ak\_percentile, param);
```
6.2.8 qos_statement

**Responsibilities**  The qos_statement object shall represent a QoS statement from CQML. Its main responsibility is thus to contain constraints on qos_characteristics.

The qos_statement must thus have a method for adding constraints on qos_characteristics. The method must allow for placing a constraint on the qos_characteristic itself, or on one of its statistical aspects.

The qos_statement must have a method for testing if it conforms to another qos_statement. This is necessary when we want to check if an object that provides the QoS specified in this qos_statement satisfies the QoS required by another object specified in some other qos_statement.

**Behaviour**

**refinement**  The qos_statement has a method for defining that it refines another qos_statement, *refines*.

**constraints**  The qos_statement has a method for adding constraints on qos_characteristics, to its set of constraints, *add_constraint*. The method takes as arguments the qos_characteristic and aspect to constrain, the relational operator and the constraining value. The method must thus be called once for each aspect of a qos_characteristic we want to constrain.

**conformance**  A very important function of the repository is to test if one qos_statement conforms to another. The *conforms_to* method is responsible for providing this functionality. The method takes as argument a reference to another qos_statement. If the qos_statement is at least as strong as the one past as a parameter the method returns true, false otherwise.

As a short example, lets say we have a qos_statement A, with the constraint delay < 5, and another qos_statement B, with the constraint delay < 6. Calling the *conforms_to* method on A, with B as argument would result in the method returning true. If the method is invoked on B, with A as argument, would return false.

For a qos_statement to test whether it is stronger than some other qos_statement, it needs to inspect the constraints of the other qos_statement. The qos_statement has thus a method for exporting its constraints as a sequence, *constraints*.

**Examples**  As an example we will insert the qos_statements from section 5.3 into the repository.

// First we create the fast qos statement,
// which is a simple constraint on the
// qc_delay characteristic

qos_statement_ptr qs_fast;
qs_fast = cat_time -> create_qos_statement("fast");

// AspectDescription is used to specify a constraint
// on a statistical aspect of the qos_characteristic

AspectDescription aspect_descr;

// We use ak_none to specify that this is a constraint
// on the qos_characteristic itself

aspect_descr.kind = ak_none;
aspect_descr.op = lt; // less than
aspect_descr.value.kind = vk_number;
aspect_descr.value.number = 10;

qs_fast -> add_constraint(qc_delay, aspect_descr);

// We now specify the not_so_fast qos_statement, which
// places constraints on two statistical aspects of the
// statistical_delay qos_characteristic

qos_statement_ptr qs_not_so_fast;
qs_not_so_fast = cat_time -> create_qos_statement("not_so_fast");

// First we specify a constraint on the mean delay

AspectDescription aspect_descr;

// We use ak_mean to specify that this is a
// constraint on the mean

aspect_descr.kind = ak_mean;
aspect_descr.op = lt;
aspect_descr.value.kind = vk_number;
aspect_descr.value.number = 20;

qs_not_so_fast -> add_constraint(qc_stat_delay, aspect_descr);

// Second we specify a constraint on the 90th percentile
aspect_descr.kind = ak_percentile;
aspect_descr.op = lt;

// The param field of the AspectDescription struct is
// used to specify the parameter to the aspect

aspect_descr.param.kind = pk_number;
aspect_descr.param.number = 90;
aspect_descr.value.kind = vk_number;
aspect_descr.value.number = 25;

qs_not_so_fast -> add_constraint(qc_stat_delay, aspect_descr);

Now that we have specified two qos_statements, we want to see how they can be used to decide on whether an object offers satisfactory QoS, or not.

Let us for instance say that we have a server object, A, that offers the qos_fast statement, and a client object, B, that requires the qs_not_so_fast statement. A negotiator whose task is to determine if object A could fulfill the requirements of object B, could use the following piece of code.

if( qs_fast -> conforms_to( qs_not_so_fast ) ) {
    // ok, B will be happy with A’s services
} else {
    // not ok, B will not be satisfied with A
}

The conformance test of the last example will evaluate to false, because the two statements have different statistical aspects, and can not be compared.

**Code generation**  One may have noticed that building and inserting QoS specifications into the repository requires some amount of code. This code is, however, easily generated from the CQML specification, and is as such output from a code generator. This is addressed by Lundby in his work [Lundby 2004].

The general idea, however, is to have a tool that is used to insert QoS specifications into the repository. The objects in the system could then simply use the QoS specifications already in the repository.
6.2.9 qos_reference

Responsibilities Because of the way you can define a QoS category in CQML, by first defining a QoS characteristic, and the defining that the characteristic should be in a given category, we need an object that can be a reference to some object that is defined elsewhere. This is the task of the qos_reference.

Behaviour The qos_reference has one method to return the object it references, get_reference. This method returns the Contained object it references.

6.3 QoS Negotiation and Monitoring

So far, the objects in the repository have not taken any parameters, and we have, thus, not been able to bind any EventSequences to the qos_characteristics or qos_statements. When it comes to negotiating and monitoring QoS, we need to specify what qos_statements we require, or support, for different interface events.

We introduce three new classes for negotiation and monitoring. The Negotiator which handles the QoS negotiation, on behalf of an object. The QoS Monitor which manages monitoring of QoS, and the qos_real class, that is a kind of qos_statement for keeping track of measured QoS.

The Negotiator and QoS Monitor objects bind qos_statements in the repository to interface entities. In this version of the QoS repository, interface entities are simply interface method calls. A qos_statement can not span multiple interface entities. It is the task of the code generation tools to map EventSequences to interface entities.

This is a simplification compared to CQML. You can for instance not specify that the delay from you invoke a method on the interface, till you receive data on a stream on the same interface, should not exceed some value. It should, however, not be a big task to expand the design to handle EventSequences instead of interface entities.

6.3.1 Negotiator

When a client object is to decide whether it should use the services of a server object or not, it must take into consideration the QoS the server object provides. It may also have to make obligations to the server object.

The process of negotiating QoS is a complex one. It can possibly involve negotiating with several server objects, to choose the one with the best QoS. Taking into account which qos_characteristics that matters the most, etc.

How well the repository objects can be used in a negotiation process, and the automatic generation of negotiators are important evaluation points
on how well CQML is suited as a QoS modelling language. The design and
evaluation of a complex negotiator, and autmagically generating a negotiator
is to big a task for this work, and will have to be included in some future
work.

For our testing, we will design a simple, and rather stupid Negotiator. It
will simply know what qos_statements an object provides, and the qos_statements
it requires, to deliver this QoS. The client and server will each have a Ne-
gotiator. The Negotiator of the client object will initiate the negotiation
with the Negotiator of the server object. The client Negotiator states what
qos_statements the client requires. If the server has any requirements on
the client, the server Negotiator will state the requirements on the client. If
both client and server can meet the requirements of the other, the result of
the negotiation will be true to indicate that the server can deliver a service
with adequate QoS.

Responsibilities  The responsibilities of a Negotiator is to keep references
to the qos_statements that an object provides, and to negotiate QoS with
Negotiators of other objects.

Behaviour

negotiate  The Negotiate interface has only one method, negotiate. This
method takes as argument the qos_statements required of the object in
question. The method is used both to initialize negotiation, and to place
requirements on the client object.

6.3.2 qos_real

Now that we have these objects for specifying QoS, we want to see if we
are getting what we required. We, thus, need an object for representing
measured QoS.

Measured QoS is a statement about the real QoS we receive, and is
as such also a qos_statement. A qos_real object is used to keep a log of
measured QoS. This information by itself could be quite useful to a system
administrator when dimensioning the system. For the system to be able to
adapt to changes in QoS at runtime, we need methods to test if the measured
QoS conforms to the QoS specified in a qos_statement.

The qos_real interface is a specialisation of the qos_statement interface,
but a qos_real object is, however, not a refinement of a qos_statement
object. When we refine a qos_statement, we can only make it stronger, so a
refined qos_statement will always conform to the qos_statement it refines.
The idea behind the qos_real object is to measure received QoS, and check
whether it satisfies the requirements as specified in some qos_statement.
Saying that a qos_real object refines a qos_statement would be the same as guaranteeing that the object will deliver the specified QoS.

qos_real and qos_statement have a lot in common, but their implementations differ in that qos_real will only have equality constraints, and they are not really constraints, but measured QoS.

Responsibilities  The object has two main responsibilities. It should represent the real QoS for some QoS characteristic of the system. Second, it should be able to check whether it conforms to some specified QoS.

The qos_real object must keep a log of the measured QoS on some of the QoS characteristics of the system. It is the responsibility of the qos_real object to calculate the values for the statistical aspects specified in the qos_characteristics it holds measurements on.

The qos_real object must be able to tell whether it conforms to some specified qos_statement.

It makes no sense for a qos_statement, representing specified QoS, to test if it conforms to a qos_real object. The only case where a qos_statement will conform to a qos_real is when the qos_statement only specify equality constraints. In this special case, the qos_statement conforms to the qos_real, if and only if the qos_real conforms to the qos_statement.

There is little use testing if a qos_real conforms to another qos_real. The only time the two would conform to each other is when they are equal. What could be interesting is to compare two qos_real objects, to see which one has the best QoS. This feature will have to wait for the next version of the repository.

For the aforementioned reasons, the qos_real object does not have to expose its constraints, as is required by the qos_statement interface. If we later add the possibility to compare qos_real objects, this would again be a necessity. A constraint on a qos_real object is the same as the measured QoS. It is an equality constraint, constraining the value of a qos_characteristic to the observed QoS.

Behaviour

measurement  The measurement method of the qos_real interface takes a qos_characteristic object, and a value on the observed QoS for the corresponding characteristic, as arguments. When the qos_real object receives a measurement, it must log the value for the specified qos_characteristic. If it has not received any measurements on the specified qos_characteristic earlier, it will have to create a log for that qos_characteristic.

The qos_real have to keep a log of the measurements to be able to calculate values for the statistical aspects. For some of the aspects it is possible to update the value for each measurement, without a need for a
log, for instance maximum, minimum and average. This is not possible for statistical aspects like percentile.

If we were to measure QoS for some characteristic over a longer period of time, the most recent measurements would make little impact on the value of some of the statistical aspects for that qos_characteristic. To get a correct measure of the current QoS, only the most recent measurements should be used when calculating the statistical aspects. This could be measurements the last minute, or maybe the last 1000 measurements. How the statistical aspects are calculated is implementation specific.

What might be a correct average for one system, might be wrong for another. One system may want the average to be for the measurements in the last minute, while the other wants the average for the entire day. It should thus be possible to specify in some way how the statistical aspects should be calculated.

**conformance** The qos_real interface inherits the *conforms_to* method from the qos_statement interface. The task of this method is to check that the observed QoS conforms to some agreed QoS, specified in a qos_statement.

**Examples** In this example we will create a qos_real object in the repository. The repository might not be the best place to keep the QoS measurements. Several things must be considered when deciding where to keep the measurements. For instance how fast the system must respond to violations on agreed QoS, or how big the load on the repository would be if there are many objects that need to be serviced.

In this example, an object defines a calculate method, and it promises that the delay from a client object calls this method till a response is emitted is less than 10 ms. That is, it offers the statement qs_fast for this method.

We thus have to make measurements on the qc_delay characteristic for this method, to see if the object lives up to its promises. To do this, we have an object with a listener method, which is called every time a response is emitted.

```cpp
public class Listener {

private:

qos_real_ptr qr_delay;
qos_characteristic_ptr qc_delay;

public:

void init(Container_ptr c, qos_characteristic_ptr qcd)
```
qr_delay = c -> create_qos_real(CORBA::nil);
qc_delay = qcd;

void calculate_response_emitted(int delay)
{
    Value value;
    value.kind = vk_number;
    value.number = delay;
    qr_delay -> measurement(qc_delay, value);
}

CORBA::Boolean conforms_to(qos_statement qs)
{
    return qr_delay -> conforms_to(qs);
}

The qr_delay should now keep a log of the delay of the calculate method of the object. It should further calculate the different statistical aspects specified by the qc_delay characteristic.

To check that the object delivers what it promised, you simply have to call the conforms_to method of the Listener object. The object in this case promised that the calculate method would offer the qs_fast statement.

Listener l;

l.init(repository, qc_delay);

// do some more...

if( l.conforms_to(qs_fast) ) {
    // all is well, the object delivers
} else {
    // QoS offer violated
}
6.3.3 qos_monitor

A monitor must do three things. It must calculate values for the different qos_characteristics when there is some interaction at an interface. Second, it must map interface entities to qos_real objects. Third, it must check that the measured QoS conforms to the required QoS, and notify the involved parties, if it does not.

There are several issues that must be taken into consideration when it comes to monitoring QoS. Where the monitor should be running depends on these issues. If the client object needs to respond quickly to a QoS contract violation, we might need to run the monitor in the client runtime space. If the client is low on resources, it might be better to keep the monitor in the repository, to avoid burdening the client.

Some pros and cons of keeping the qos_monitor in the repository or in client space.

**Repository pros**

- Less code in the application for handling monitoring. Makes it less platform/language dependent
- Faster to get started with QoS
- Frees up time in the client/server to handle business logic

**Repository cons**

- Will take more time to react to QoS violations
- Mocks up network with QoS-events

**Client space pros**

- Can react to QoS violations more quickly
- Does not mock up network with QoS-events

**Client space cons**

- Needs code to run in client space. The qos_monitor must support the platform the client is running on
- QoS monitoring will eat CPU cycles that could be used to handle business logic
As you can see, the pros of one solution is the cons the other. Which solution is best, will vary from case to case, and we will need to support both. We specify an interface for the qos_monitor, and the implementing object is free to live wherever it may please.

There are, however, some code that needs to live at the interface it is monitoring. This code could be sending raw events to the qos_monitor, corresponding to events in the SE and SR EventSequences of the interface. Another solution is to have this client space code calculate values for the qos_characteristics that are monitored, and send these values to the qos_monitor.

We choose the latter solution. With this solution we can create a generic qos_monitor that will work for all cases. The code that calculates the values for the qos_characteristics can be generated from CQML.

The code that calculates the values for the qos_characteristics and reports to the qos_monitor will typically be living in a binding object, or a proxy for the server object.

Responsibilities

The qos_monitor should have a map that maps interface entities to qos_statements. It should also have a corresponding qos_real object for each qos_statement. When a measured value for a qos_characteristic is received, it should be directed to the correct qos_real object.

The qos_monitor is also responsible for checking that the measured QoS conforms to the negotiated QoS, and notify the interested parties if it does not conform.

Behaviour

contract

We need a way to tell the qos_monitor about the contract it is going to monitor. We do this with the register_statement method. This method takes as argument a reference to a qos_statement, and the name of the interface entity to monitor. This method must be called once for each qos_statement in the contract.

measurement

When there has been some interaction at the monitored interface, we need methods to report these measurements. We do this with the measurement method. It takes as argument, the name of the interface entity, a reference to the qos_characteristic, and the value.

listeners

We also need a method where interested parties can register themselves, to receive notification when there is a QoS violation.

Example

For an example of how the qos_monitor is supposed to be used, take a look at figure 6.3
6.4 CQML Parser

We now have a language for specifying QoS aspects of distributed systems, and a way to represent and manipulate QoS information at runtime and design time. The next step is to transfer the information specified with CQML into objects that can live in the QoS repository. With this information in the repository, it can be used as both a design time and a runtime tool. We would also like to create runtime tools, like negotiators and monitors from the CQML and object language specifications.

This poses many challenges. First, we need to make CQML work with some modelling language, or object definition language. It would be best if these two languages were orthogonal, but there have to be a few interdependencies, like specifying requires and provides clauses, and symbol resolution.

Second, the task of transforming a CQML specification to something that is useful at runtime is not a straightforward task itself. First, the CQML specification must be parsed. We have good tools to help us accomplish this. After parsing, there are several possibilities. We could insert the QoS objects directly into a QoS repository. We could make code that communicates with a QoS repository to insert and use the objects at runtime, including negotiators and monitors.

Yet another scenario arises when we consider using the repository as a design tool. In this scenario, not only must the parser use the CQML specification for resolving symbols, but it must also look up design objects in the repository.

We divide the design of the parser in three. We have the implementation of the symbol table, that should be shared between the object language and the CQML. Second, we have the parser itself, that parses CQML and some object languages, and builds up a datamodel representation of the definitions, a parse tree. Last, there is the emitter, that uses the parse tree to populate the repository, or generate runtime code.

We make our task a little easier, by not using the repository during parsing. The parser does not populate the repository directly, but creates code that populates it. The repository is not used for looking up symbols during parsing. This simplification saves us a good deal of work, and is not important in evaluating the implementation of the CQML runtime.

6.4.1 Object language and symbol resolution

The purpose of CQML is to be used together with a modelling language, to fill in what the modelling language lacks in terms of QoS specification. The task of the modelling language to specify the functional aspects of the system is almost orthogonal to the task of CQML to specify the QoS of the system.

We use the modelling language to specify what objects should live in the
system, and how they communicate. We use CQML to specify QoS characteristics and QoS statements. Both these tasks can be done independent of one another.

At some point we need to connect the two languages, for there to be any point in all this madness. To keep the work of introducing CQML into an existing modelling language to a minimum, we only let the languages interact in specifications of requires and provides clauses, and in contracts.

This makes it possible to do the system design in a modular fashion, and opens up for a modular design of the design tools.

If we limit the QoS characteristics and statements to work with events and eventsequences, the specification of these QoS objects does not depend on anything from the modelling language. The qos_characteristic, qos_statement and qos_category objects can thus be specified separately. They can further be handled by a separate module.

Depending on the modelling language and the modelling tools in use, there are two possible ways to introduce the provides and requires clauses. You can expand the construct for specifying objects in the modelling language itself by introducing a construct to specify requires and provides clauses. Another is to separate the provides and requires clauses from the modelling language by creating new constructs for specifying that object A provides qos_statement B, and so on.

Either way, the module for handling the specifications of provides and requires clauses, and contracts, must do a look up of the symbols representing qos_statements. The best way to accomplish this is to create a component for looking up the qos_statement based on the symbols in the provides and requires constructs of the object specification. This component can be implemented to either parse CQML specifications from file, or use the QoS repository to look up the qos_statements. In this way, the modelling language tools do not have to burden themselves with understanding most of the CQML syntax.

6.4.2 Symbol table

In this parser experiment, we will try to incorporate a CQML parser into a potentially existing object language parser. Instead of searching for an open source object language parser, we implement a very simple TINA-ODL parser, that does as little as possible.

An important part of a parser is the symbol table, so that we can look up definitions and check for type safety. In this design, we try to create a simple interface to a symbol table. We design it based on the needs of the CQML parser. The idea is that it should be feasible to integrate this interface with the symbol table in an existing object language parser.

The interface to the symbol table is listed below. It has methods for adding symbols, and for nesting symbol tables. This way we can have nested
definitions, and we can check that a symbol is visible from the point of a
definition. If it is visible, we say that it is within scope. There are methods
to extend the set of visible symbols, through inheritance. Last, there is of
course methods to look up symbols.

```java
public interface SymbolTableEntry {

    // add a symbol at this level in the symbol tree
    public SymbolTableEntry addNewEntry(String name, SimpleNode sn);

    // add a level to the symbol tree
    public SymbolTableEntry addNewTable(String name, SimpleNode sn);

    // adding symbols from an inherited scope
    public void addExtension(SymbolTableEntry se);

    // look up a symbol at this level only
    public SymbolTableEntry findInTable(String name);

    // look up a symbol that is in scope
    public SymbolTableEntry findInScope(String name);

    // check if the symbol is defined at this level
    public boolean inTable(String name);

    // some helpers for navigating the tree
    public SymbolTableEntry getEntry(String name);
    public SymbolTableEntry getRoot();
    public SymbolTableEntry getParent();
    public SimpleNode getCreator();
}
```

6.4.3 Parse tree

The parse tree is designed to separate the parser from the emitter. With a
good data model in place, we can replace the implementation of the parser
with another, without needing to do any changes to the emitters. More
importantly, we can have different emitters that share the same parser. With
this approach, if we need to make changes to the parser, we only need to do
it at one place in the code, and not once for each emitter. We can also parse
once, and then run multiple emitters on the same parse tree, saving CPU
time.

The parse tree is illustrated in figure 6.5. It has been defined as a hier-
archy of interfaces. This model will at the top level reflect the object model
of the repository. At more detailed levels, it will correspond to objects in the
implementation of the repository. There is not a strict one to one mapping,
but it is natural that the parse tree, and the internal data representation of
the repository are somewhat similar.
Figure 6.5: The parse tree
Figure 6.6: Domain
In figure 6.6 we see that a domain can be either numeric, enumerated or set. For both enumerated and set, we have an element set, which contains a set of strings, representing the individual elements. Both of them also have an order. The order is represented as a set of (low, high) pairs.

Figure 6.7 shows that the statistical aspect can be simple or parameterized. The parameter can be a number, as for percentile, or a range.

The constraints in a qos_statement are represented by the model in figure 6.8. As we can see, a constraint consists of a statistical aspect, a relational operator and a value.

6.4.4 Emitter

When the parser has done its job, and built a nice parse tree, we can start one or more emitters, that start at the Start node, and work its way down the tree, emitting code, populating the repository, or doing other things.

An emitter must implement the Emitter interface. This interface has methods for handling definitions that are near the root of the parse tree. Some of these definitions are defined within other definitions. One kind of definition might be used in different definitions, but should be handled similarly.

When the parser is done, we invoke the emit method on the emitter. This method is responsible for handling the Start node, and will delegate
the handling of child nodes to the other methods of the emitter.

The Emitter interface is defined as an abstract class, where `emit` is the only implemented method.

```java
public abstract class Emitter {
    public abstract void emit_qos_characteristic(Iqos_characteristic node);
    public abstract void emit_qos_formal_parameter(Iqos_formal_parameter node);
    public abstract void emit_qos_domain(Iqos_domain node);
    public abstract void emit_qos_statistical_aspect(Iqos_statistical_aspect node);
    public abstract void emit_qos_simple_aspect(Iqos_simple_aspect node);
    public abstract void emit_qos_parameterized_aspect(Iqos_parameterized_aspect node);
    public abstract void emit_qos_range(Iqos_range node);
    public abstract void emit_qos_statement(Iqos_statement node);
    public abstract void emit_qos_constraint(Iqos_constraint node);
    public abstract void emit_qos_constr_aspect(Iqos_constr_aspect node);
    public abstract void emit_qos_constr_value(Iqos_constr_value node);
    public abstract void emit_qos_category(Iqos_category node);
    public abstract void emit_qos_reference(Iqos_reference node);
}
```
The `emit` method will do some initializing, then it will call the appropriate methods for each of the the qos_characteristic, qos_statement, qos_category and qos_reference nodes. The emit method will not call any of the other methods on the interface. It is up to the methods for a node to call the methods for handling the child nodes.

6.4.5 Populating and using the repository

As already stated, we will not make an emitter for populating the repository directly. We will instead make an emitter that emits C++ code for populating the repository. The generated C++ classes can also be used as utility classes during runtime.

Before we use the emitter to generate any code, we design a framework, that the generated code can build on. This framework consists mainly of classes that wrap the CORBA skeletons. In this way, we hide most of the CORBA specific code. This makes the generated code simpler, the application programmer does not necessarily have to write CORBA code, and the parser/emitter can easily be moved to another platform.

The classes that make up this framework are simple wrappers, and they correspond to the interfaces of the repository. They are simply proxies, and a detailed description of them seems unnecessary.

The generated classes use these framework classes as super classes. They add structure, and a method for creating the corresponding object in the repository.

If there are QoS constructs defined within a container, we define the corresponding class as an inner class of that container. The container would typically be a qos_category, or the repository itself.

For an example of code generated by the emitter, have a look at section 7.3. The classes that make up this framework are defined in appendix C.
Chapter 7

Implementation

7.1 Repository

With the simplifications made in the design, implementing the repository was not very challenging. There are a few things worth noting.

7.1.1 Lifecycle

Using methods for adding features to the specificatinos in the repository works well, but there is no method to say that the object is finished, and ready to be used. There should be an attribute on the QRObj ect stating whether an object is in an initial or active state.

7.1.2 qos_characteristic equality

The idea with CQML is that qos_characteristics should be matched on equal semantics. Because semantics are not represented in the repository, the has_equal_semantics method will have to be implemented with simple object reference equality. That is, we check that the argument to the method references this, the object itself.

7.1.3 qos_statement conformance

To check that one qos_statement conforms to another is quite complex. Say for instance that we have two qos_statements, A and B. If we want to check if A conforms to B, we call

A -> conforms_to( B );

For each constrained qos_characteristic in B, we must find the corresponding constraint in A, and check that it is at least as strong as the one in B.
Because we do not support parameters or semantics on qos_characteristics, the task of finding the corresponding constraints, is simply finding the constraints that references the same qos_characteristic. For doing this, we use the has_equal_semantics on the qos_characteristic.

A qos_statement can have constraints on several statistical aspects for each qos_characteristic. We must check that for all statistical aspects that are constrained in B, there is a corresponding constraint in A, and that the constraint in A is as strong as the one in B, or stronger.

We will now take a closer look at the implementation of the conforms_to method.

```c++
CORBA::Boolean qos_statement_impl::conforms_to(CQML::qos_statement_ptr qs)
  throw(CORBA::SystemException)
{
  CQML::relOp op, cpop;
  CQML::qos_characteristic_var qc;
  Aspect *asp;
  Constraint *cp;
  CQML::ConstraintDescriptionSeq *cd;

  cd = qs->constraints();
  CQML::ConstraintDescriptionSeq &cd_var = *cd;

  // for each characteristic constrained by statement
  for(int i=0; i<cd_var.length(); i++){
    qc = cd_var[i].characteristic;
    // find the constraints for the characteristic
    cp = _constraints;
    while(cp != NULL){
      if( cp->characteristic->has_equal_semantics(qc) )
        break;
      cp = cp->next;
    }
    if(cp == NULL) return false;

    // for every aspect specified for the characteristic
    for(int j=0; j<cd_var[i].aspects.length(); j++){
      CQML::AspectDescription &wk_asp = cd_var[i].aspects[j];

      // must find corresponding aspect
      asp = cp->aspects;
      while(asp != NULL){
        if(asp->ad.kind == wk_asp.kind){
          if(wk_asp.kind != CQML::ak_percentile &
            wk_asp.kind != CQML::ak_moment &
            wk_asp.kind != CQML::ak_frequency)
            break;
          if(equals(asp->ad.param, wk_asp.param))
```
This method uses a helper method to check for conformance on statistical aspect level, the `conforms` method. This method is listed below.

```c++
bool qos_statement_impl::conforms(CQML::qos_characteristic_ptr qc,
                                  const CQML::AspectDescription& strong,
                                  const CQML::AspectDescription& weak)
{
    int comp;
    CQML::relOp sop, wop;

    sop = strong.op;
    wop = weak.op;

    comp = qc->compare(strong.val, weak.val);

    switch(wop){
    case CQML::eq:
        if((sop == CQML::eq && comp > 0) ||
           (sop == CQML::lt && comp >= 0) ||
           (sop == CQML::le && comp > 0) ||
           (sop == CQML::gt && comp >= 0) ||
           (sop == CQML::ge && comp > 0))
            return true;
        break;
    case CQML::ne:
        if((sop != CQML::ne && comp >= 0) ||
           (sop == CQML::lt && comp >= 0) ||
           (sop == CQML::le && comp > 0) ||
           (sop == CQML::gt && comp >= 0) ||
           (sop == CQML::ge && comp > 0))
            return true;
        break;
    case CQML::lt:
        if((sop == CQML::eq && comp > 0) ||
           (sop == CQML::lt && comp >= 0) ||
           (sop == CQML::le && comp > 0))
            return true;
        break;
    case CQML::gt:
        if((sop == CQML::eq && comp > 0) ||
           (sop == CQML::gt && comp >= 0) ||
           (sop == CQML::ge && comp > 0))
            return true;
        break;
    case CQML::ge:
        if((sop == CQML::eq && comp > 0) ||
           (sop == CQML::ge && comp >= 0) ||
           (sop == CQML::gt && comp >= 0) ||
           (sop == CQML::lt && comp >= 0) ||
           (sop == CQML::le && comp > 0))
            return true;
        break;
    case CQML::le:
        if((sop == CQML::eq && comp >= 0) ||
           (sop == CQML::ge && comp >= 0) ||
           (sop == CQML::gt && comp >= 0) ||
           (sop == CQML::le && comp > 0))
            return true;
        break;
    default:
        return false;
    }
}
```
(sop == CQML::le & comp & comp > 0))
    return true;
  break;
  case CQML::le:
    if((sop == CQML::eq || sop == CQML::lt
        || sop == CQML::le) && comp >= 0)
      return true;
    break;
  case CQML::gt:
    if((sop == CQML::eq && comp > 0) ||
        (sop == CQML::gt && comp >= 0) ||
        (sop == CQML::ge && comp > 0))
      return true;
    break;
  case CQML::ge:
    if((sop == CQML::eq || sop == CQML::gt
        || sop == CQML::ge) && comp >= 0)
      return true;
    break;
  }
}

return false;

This method checks that the strong argument conforms to the weak argument. That is, that the constraint represented by the strong argument, in fact is stronger than the constraint represented by weak.

The first thing we do, is to compare the values of the two constraints. The compare method on the qos_characteristic takes into account the ordering of the domain, so that if the first argument is stronger than the second, it will result in a positive integer.

We then check that the strong constraint has a stronger combination of operator and value than the weak constraint, or as least as strong. This comparison is rather complex, and should be subject to heavy unit testing.

There is one shortcoming with this implementation. A simple example should clarify this. We will use CQML for this example.

qos_characteristic delay {
  domain: decreasing numeric ms;
  mean;
  maximum;
};

qos fast {
  delay.mean < 20;
qos rt_fast {
    delay.maximum < 10;
};

We have a statement fast saying that the mean delay should be less than 20 ms, and a real time statement, rt_fast that states that the delay must never exceed 10 ms.

If we have a runtime representation of these statements, and checks that rt_fast conforms to fast,

rt_fast -> conforms_to( fast );

the result will be false. To see why this is, we have to look at the inner loop of the conforms_to method. In this loop, we try to match constraints on statistical aspects, one to one. That is, when we have a constraint on the mean aspect in fast, we try to find a mean aspect in rt_fast. Failing to find this, the conforms_to method gets confused, and returns false.

The fact that rt_fast has a constraint that delay.maximum is less than 10, implies that delay.mean also must be less than 10. The rt_fast statement, thus, conforms to the fast statement.

### 7.1.4 CORBA references

In the design of this version of the repository, there is no requirement that there should be support for two or more repositories working together. There is no support for copying or moving QoS definitions between repositories. The CQML parser, or the code it generates, can of course be used to insert the same definitions into several repositories.

The implementation of the objects in the repository has, however, been done, so that all interchange of information between the objects are done through methods defined in the interfaces, defined in IDL.

A qos_statement can place constraints on a qos_characteristic regardless of what repository they live in. It can also check if it conforms to a qos_statement in another repository. This fact should make the task of extending the implementation to support cooperating repositories quite feasible.

### 7.1.5 Not yet implemented

The qos_real and qos_monitor classes has not yet been implemented in the repository. This should not be crucial to showing the success of this implementation of a runtime representation of CQML.
qos_real are very similar to qos_statement, and their implementations should be very much alike. Because qos_real represent measured QoS, its "constraints" will have equality operators only, making the conformance check less complex. The qos_real object is created based on a qos_statement, and will have the same statistical aspects as the statement it should conform to.

The qos_real will have to calculate values for the different statistical aspects. This can be challenging to implement, but should by no means be impossible. There exists numerous implementations of such algorithms.

The task of the qos_monitor is to direct QoS measurements to the correct qos_real objects, and check that the qos_real objects conforms to the corresponding qos_statement. Not a very challenging task.

The part of the monitor that have to live at the client or server interfaces, to observe the QoS, should ideally be generated by some object definition language parser. For now, this code will have to be hand coded. This code is, however not a part of the generic framework, but will be different from case to case.

The objects in the repository are not persistent, and will die when the repository is shut down or restarted. This is not important for evaluating the repository against the requirements in section 3.2.

7.2 Parser

In this section we describe the implementation of the parser. As noted in the design, we choose to implement a simple TINA-ODL parser, that we use to integrate the CQML parser with.

The TINA-ODL parser parses the constructs we need in CQML. There is no code generated from the TINA-ODL interface and object definitions. The parsed object definitions are simply used for requires and provides clauses.

7.2.1 CQML parser

We implemented the CQML parser, and the TINA-ODL parser in JavaCC. With JavaCC, the implementation of a parser for a language with a well defined grammar is a real joy. The CQML grammar is a good fit for JavaCC, and was indeed a joy to implement. Even though there was little trouble implementing the parser itself, building the parse tree and resolving symbols had their challenges, but not more than expected.

We designed the parse tree with interfaces. This means we will have to implement all of these interfaces. During the implementation of the parser, it became obvious that this design led to a good deal of extra code, and work.

The reason for using interfaces in the design of the parse tree was to make the parser and emitter independent of each other. The parse tree is really
nothing else than a data model. Data models are normally not designed with interfaces for each data object.

The data model should rather have been designed to best represent the constructs of the CQML language. As noted in the design, this data model will be very similar to the internal data model of the repository. With this data model in place, we could have different implementations of the parser build up a parse tree using this generic data model, independent of the emitters using the parse tree to emit code, etc.

This is the same argumentation that was used in the design to use interfaces, but a good generic data model should not have to be hidden behind a lot of interfaces. The use of interfaces in the design of the data model was not good design, and adds little else than complexity to the implementation.

7.2.2 CQML and the object language

The implementation of a parser that parses both TINA-ODL and CQML did not give rise to any problems. The grammars of TINA-ODL and CQML do not seem to have any conflicting constructs. Even though the implementation was straightforward, there are some notes to be made.

The first thing we must be aware of, is that the TINA-ODL parser implements the bare minimums of what is required from a CQML point of view. The parsing of the ODL definitions does not result in any code being generated, and the symbol table used was designed specifically with CQML in mind.

From this simple implementation of a TINA-ODL parser, we do not know if the symbol table is suited for the needs of a TINA-ODL parser, besides simple type checking. The parser has only been tested with simple object definitions, and more complex ODL definitions can cause trouble for our simple parser.

The TINA-ODL and CQML parsers were implemented as independent modules. The only common points are the symbol table, and the requires and provides clauses of the ODL object definition.

7.2.3 Emitter

The QREmitter is the only implementation of the Emitter interface made in this work. This emitter generates C++ code for populating the repository.

The emit methods for qos_characteristic, qos_statement and qos_category are implemented to be independent of one another. If, for instance, a qos_characteristic is defined within a qos_category, the emit method for the qos_category can make a call to the emit method for the qos_characteristic. This will result in an inner class being generated for the qos_characteristic, within the class of the qos_category.
With this approach, you can even nest qos_categories within each other. This way, qos_categories can function as a kind of namespace, and a directory like structure can be built. The emit methods are kept simple with this approach, but the generated code can be quite complex if there is a lot of nesting of qos_categories.

Many of the methods of the Emitter interface are only used from within the QREmitter. The design of the interface was also changed during the implementation of the QREmitter. Some of the methods that were added are probably only useful to the QREmitter, and should rather have been defined as helper methods in the QREmitter itself.

The only method used from outside the QREmitter is actually the emit() method. A revisit of the implementation of the QREmitter shows that there are more methods in the interface than are practical. A generic implementation of the Emitter might be better with an interface with methods for emitting qos_characteristic, qos_statement and qos_category only. The generic Emitter could implement a helper method that calls these methods for each definition within a container.

7.3 Test case

To show that the parser and repository plays well together, we include a simple, and maybe a bit silly example. The CQML that will be parsed, and have its objects inserted in the repository is listed below.

```plaintext
interface potato {
  server short grow();
};

interface potato_client {
  client short grow();
};

qos_characteristic delay(op : Operation) {
  domain: decreasing numeric;
}

qos_characteristic taste {
  domain: increasing enum {dirt, nothing, potato, good}
  with order {dirt<nothing, nothing<potato, potato<good};
}

qos fast_growing(op : Operation) {
  delay(op) < 100;
}

qos early_harvest(op : Operation) {
  delay(op) <= 120;
}

qos tasty {
  taste >> potato;
}
```
object beate {
  supports potato;
  provides fast_growing(potato::grow) and tasty;
};

object farmer {
  supports potato_client;
  requires early_harvest(potato_client::grow) and tasty;
};

When this CQML is run through the parser and emitter, we end up with the C++ code listed below.

```cpp
#include "QosRuntime.h"
#include "QoSLib.h"

class QRepository : virtual public qr_base 
{
public:

  QRepository(QoSRepository_ptr qr) {
    ref(qr);
    init();
  }

  QRepository &repository() {
    return *this;
  }

public:
  class qc_delay : virtual public qc_base 
  {
    protected:
      QRepository* _container;
    public:

      void container(QRepository &cont) {
        _container = &cont;
      }

      QRepository &container() {
        return _container;
      }

      QRepository &repository() {
        return _container->repository();
      }

    public:

      void create() {
        cout << "creating delay" << endl;
        qos_characteristic_ptr qc;
```
try {
    qc = container().create_qos_characteristic("delay", decr_numeric);
    ref(qc);
    } catch(NameCrash &e) {
        cerr << "Name crash: delay" << endl;
    } };

cqc_delay delay;

class qc_taste :
    virtual public qc_base
{
    protected:
        QRepository* _container;

    public:
        void container(QRepository &cont) {
            _container = &cont;
        }
        QRepository &container() {
            return *_container;
        }
        QRepository &repository() {
            return _container->repository();
        }

    public:
        void create() {
            cout << "creating taste" << endl;
            qos_characteristic_ptr qc;
            try {
                qc = container().create_qos_characteristic("taste", ordered_incr_enum);

            IdentifierSeq elements;
            elements.length(4);
            elements[0] = (const char*)"dirt";
            elements[1] = (const char*)"nothing";
            elements[2] = (const char*)"potato";
            elements[3] = (const char*)"good";
            qc->domain_values(elements);

            ValueOrderSeq order;
            order.length(3);
            order[0].low = (const char*)"dirt";
            order[0].high = (const char*)"nothing";
            order[1].low = (const char*)"nothing";
            order[1].high = (const char*)"potato";

            qc_delay delay;
order[2].low = (const char*)"potato";
order[2].high = (const char*)"good";
qcs->domain_order(order);
ref(qc);
} catch(NameCrash &e) {
    cerr << "Name crash: taste" << endl;
}
);
qc_taste taste;

class qs_fast_growing :
    virtual public qs_base
{
    protected:
        QRepository* _container;

    public:
        void container(QRepository &cont) {
            _container = &cont;
        }
        QRepository &container() {
            return *container;
        }
        QRepository &repository() {
            return _container->repository();
        }

    public:
        void create() {
            cout << "creating fast_growing" << endl;
            qos_statement_ptr qs;
            try {
                qs = container().create_qos_statement("fast_growing");
                ref(qs);

                AspectDescription aspect_descr;
                AspectParam &param = aspect_descr.param;
                Value &value = aspect_descr.value;
                aspect_descr.kind = ak_none;
                aspect_descr.op = lt;
                value.kind = vk_number;
                value.number = 100;
                qs->add_constraint(repository().delay.ref(), aspect_descr);
            } catch(NameCrash &e) {
                cerr << "Name crash: fast_growing" << endl;
            }
        }
    }
```cpp
};
qs_fast_growing fast_growing;

class qs_early_harvest :
  virtual public qs_base
{
protected:
  QRepository* _container;

public:

  void container(QRepository &cont) {
    _container = &cont;
  }

  QRepository &container() {
    return *_container;
  }

  QRepository &repository() {
    return _container->repository();
  }

public:
  void create() {
    cout << "creating early_harvest" << endl;
    qos_statement_ptr qs;
    try {
      qs = container().create_qos_statement("early_harvest");
      ref(qs);

      AspectDescription aspect_descr;
      AspectParam &param = aspect_descr.param;
      Value &value = aspect_descr.value;

      aspect_descr.kind = ak_none;
      aspect_descr.op = le;
      value.kind = vk_number;
      value.number = 120;
      qs->add_constraint(repository().delay.ref(), aspect_descr);
    } catch(NameCrash &e) {
      cerr << "Name crash: early_harvest" << endl;
    }
  }
};
qs_early_harvest early_harvest;

class qs_tasty :
  virtual public qs_base
{
protected:
```
QRepository* _container;

public:

void container(QRepository &cont) {
   _container = &cont;
}

QRepository &container() {
   return *container;
}

QRepository &repository() {
   return _container->repository();
}

public:

void create() {
   cout << "creating tasty" << endl;
   qos_statement_ptr qs;

   try {
      qs = container().create_qos_statement("tasty");
      ref(qs);

      AspectDescription aspect_descr;
      AspectParam &param = aspect_descr.param;
      Value &value = aspect_descr.value;

      aspect_descr.kind = ak_none;
      aspect_descr.op = ge;
      value.kind = vk_element;
      value.element = (const char*)"potato";
      qs->add_constraint(repository().taste.ref(), aspect_descr);
      } catch(NameCrash &e) {
      cerr << "Name crash: tasty" << endl;
      }
   }
}

qs_tasty tasty;

void init() {
   delay.container(*this);
   taste.container(*this);
   fast_growing.container(*this);
   early_harvest.container(*this);
   tasty.container(*this);
}

void create_all() {
   delay.ref();
   taste.ref();
   fast_growing.ref();
   early_harvest.ref();
   tasty.ref();
}
This code can be included in the code of the distributed application, and
compiles and runs without any problems. The result is that the specified
qos_characteristics and qos_statements are inserted in the repository. The
generated code can also be used as a proxy to the objects in the repository,
to test for conformance and so forth.

There has also been other test cases, to test other domains and con-
straints on aspects, but no complete test on all variations. The example in
this section shows that the repository and parser works, but not that they
are bug free.
Chapter 8

Evaluation

In this chapter we will evaluate the repository, CQML and parser against the requirements set forth in section 3.

8.1 Repository

In this section we evaluate each of the requirements in section 3.2

8.1.1 Concepts

Does the repository support the same concepts as the modeling language?

Table 8.1 shows what concepts from CQML that is represented in the runtime repository. For all the concepts where the answer is NO, it is so because it has not been implementet in the repository yet. In the case of the provides, supports and qos_contract, it is not obvious that it should be part of the repository, but rather result in parser generated code for negotiation and monitoring of QoS.

Table 8.1 shows that the implementation corresponds well to CQML, but that it is not a complete implementation. There is still work to be done.

8.1.2 Overhead

Does the repository impose a big overhead?

There has not been run any tests to measure the time used in the different phases of QoS management in a real life like environment.

The application might suffer from longer startup times because of the QoS negotiation. We will focus on the overhead during execution of the application.

The design of the repository is flexible when it comes to where the QoS objects should live and execute. Part of the QoS monitor can run in the repository, on the client side of the end user application or somewhere else completely.
Table 8.1: Supported concepts from CQML

<table>
<thead>
<tr>
<th>qos_characteristic</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric domain</td>
<td>YES</td>
</tr>
<tr>
<td>Enumeration domain</td>
<td>YES</td>
</tr>
<tr>
<td>Set domain</td>
<td>YES</td>
</tr>
<tr>
<td>Domain order</td>
<td>YES</td>
</tr>
<tr>
<td>Units</td>
<td>NO</td>
</tr>
<tr>
<td>Statistical aspects</td>
<td>YES</td>
</tr>
<tr>
<td>Refinement</td>
<td>YES</td>
</tr>
<tr>
<td>Parameters</td>
<td>NO</td>
</tr>
<tr>
<td>Semantics</td>
<td>NO</td>
</tr>
<tr>
<td>qos_category</td>
<td>YES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>qos_statement</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistical aspects</td>
<td>YES</td>
</tr>
<tr>
<td>Refinement</td>
<td>YES</td>
</tr>
<tr>
<td>Parameters</td>
<td>NO</td>
</tr>
<tr>
<td>provides and supports</td>
<td>NO</td>
</tr>
<tr>
<td>qos_contract</td>
<td>NO</td>
</tr>
</tbody>
</table>

qos_characteristic, qos_statement, qos_category and qos_real objects are meant to be living in the repository, but can theoretically live anywhere on the object bus, like any CORBA object. Depending on where we physically place the repository, the reporting of measured QoS to the repository and the conformance checks will introduce some overhead on the network resources, or on the CPU on the client or server side of the application.

The qos_monitor is implemented to do the calculation of measured QoS at the interfaces of the application, and do the book keeping in the repository. The calculation of measured QoS depends on the quality of the code generator, and can also be hand tuned from case to case. The calculation of measured QoS can be done on another computer, saving CPU time at the interfaces, but requiring more of the network resources.

We can conclude that the repository, or runtime system, is flexible, and can be configured to use little of the resources that are scarce. With the incompleteness of the implementation, and the lack of benchmarks, we can not say anything about how suited it is for use in a real time application.

8.1.3 Platform independence

Designing and implementing the repository on top of CORBA ensures that it can be run on several platforms. It can also be accessed from applications written in any language supported by CORBA.

The use of C++ and the GNU compiler makes it possible to compile the
repository for running on several architectures and platforms. Both C++
and the GNU compiler is widely supported.

Even though CORBA is a popular framework for distributed applica-
tions, there are other options, for instance Java Enterprise Edition and .NET. The
interface is quite simple, and should be easy to translate to any of Java
EE and .NET. C++ can be incorporated in both frameworks, and it should
be possible to reuse much of the implementation. The implementation of the
core of the repository could have been better separated from the CORBA
stubs. This would have made it easier to wrap another implementation
around it.

In the future there will probably be other frameworks to support, but
then CQML will probably also have evolved, and new implementations are
called for anyhow.

8.1.4 Repository interaction

The distributed application must communicate with the repository during
negotiation and monitoring. It must also have callbacks to handle QoS con-
tract violations.

It is desirable for the negotiator to be automagically generated. The
application will then only have to initiate the negotiation. The implement-
ation of the parser does not automatically generate negotiators, so they will
have to be hand coded. It should be possible to keep this code in a separate
module from the business logic.

The monitor is also supposed to be automatically generated. The gen-
erated code will hide the interaction with the repository. This code is best
placed in a proxy on the interfaces.

The business logic of the distributed application will not be cluttered
by QoS managing code with this implementation. The application must do
some initiation of the QoS framework, it must listen for QoS violation events.

8.1.5 Design time vs runtime specifications

Can we define QoS specifications at both design time and runtime?

Of course, the CQML language is used to define QoS specifications at
design time. The parser does not have support for inserting the specifications
directly into the repository, but it generates code to insert the specifications
into the repository. The same functionality that is used by this automatically
generated code can be used to insert specifications that has not been defined
at design time.

The repository can also be used to look up already defined QoS specifi-
cations in the repository from some design tool. Objects in the repository are
not editable, and the repository is not suited to store work in progress. The
repository has only methods for adding features to the QoS specifications, not editing the ones already added.

8.1.6 Anonymous QoS objects

Does the repository support the creation and use of anonymous objects in the repository?

Yes, it does.

8.1.7 Parameters

Does the repository support parameters on the QoS specifications?

No, parameter support is not implemented in this version of the repository. This is left for a future version of the repository.

8.1.8 QoS monitoring

Does the repository have support for representing measured QoS and monitor that the real QoS is according to the negotiated QoS contract?

The repository has been designed with functionality for monitoring QoS, but it has not yet been implemented. This design allows us to monitor the value of qos_characteristics associated with some entity on the interface, and check that it conforms to the constraints in some required qos_statement.

In CQML we can specify constraints that are more general, and can have constraints that span more than one interface entity. Before parameters are implemented in the repository, this will be a shortcoming compared to CQML.

8.2 CQML and code generation

8.2.1 Parsing

Is CQML well suited for parsing?

As we have shown, in section 7.2.1, the parser for CQML was implemented in JavaCC, and the input files are parsed in a single sweep. The parser does not have to look further than the next token, to decide what element it is handling.

CQML is thus, well suited for parsing.

The symbol table proved to be a challenging task to design and implement. The task of implementing the symbol table and the uncertainty of its suitability to be integrated in an existing parser, suggests that we should seek other solutions. A better approach might be to instead of having a symbol table, have the parser, or emitter, insert QoS objects into the repository
directly. The repository would thus have the responsibility of verifying that the referenced objects exists.

8.2.2 Generated code

Can the code generated from parsing a CQML file be used to populate the repository?

In section 7.3 we parsed a simple CQML specification, and used the generated code to insert the qos_characteristics and qos_statements into the repository. Even though the example is simple, it shows that the parser and repository plays together, and delivers what is expected.

More thorough testing will of course expose several bugs, more or less serious, but this is the case of any software system.

Can the CQML specification be used to automatically generate negotiators and monitors?

The combination of parameters on qos_characteristics and qos_statements, semantics and provides and requires clauses, should be enough to automatically generate code that can be used in QoS negotiation and monitoring. We might not be able to generate code that in some kind of magic way can do all the QoS setup and monitoring, but it should be feasible to generate components that can be used in the negotiation and monitoring process.

Generating code for negotiation and monitoring has not been the focus of this work, so this is merely some thoughts on the subject.

8.2.3 Adaptation

Does CQML support specifying how to adapt to variations on QoS?

There are no constructs in CQML to specify what action to take if the QoS drops below what has been agreed on. Neither does it have support for setting higher priorities on some QoS than others.

8.2.4 CQML constructs

Is it possible to combine QoS specifications to form new, more complex QoS specifications?

CQML does not support combining QoS specification in sequence, for example saying that a qos_statement represents the overall time used to call a method, as a sequence of the delay of the network communication, and the execution time of the method on the remote server object.

To summarize, CQML evaluates well against the requirements from section 3.1, but it lacks a few features that would be nice to have.
Chapter 9

Conclusion and Future work

9.1 Conclusion

The goals of this work was to show that we can create a runtime representation of the CQML constructs, and to create a parser that can populate the repository and generate code for use at runtime.

9.1.1 Runtime representation

We have seen that the implementation of the repository supports specification of qos_characteristics, qos_statements, qos_categories and references. These objects live in the repository, and can be used to check whether a qos_statement conforms to another or not.

The runtime repository uses the same concepts as CQML, and the objects has simple methods for building the QoS objects. Someone who knows CQML, should have no trouble understanding what the runtime code does.

The representation of measured QoS has not been implemented, but the design is very near to that of the qos_statement, so an implementation of the qos_real class should be a feasible task.

There are some features of CQML that has not been implemented. One major and important feature is the representation of semantics on qos_characteristics. This is a major task, and could be an individual piece of work. Another unimplemented feature is parameters on the QoS specifications.

There are three reasons that leads us to conclude that it would be worthwhile to do some further work on the runtime repository.

- The successful implementation of the core concepts of CQML.
- The demonstration of a working repository in section 7.3.
- The reasoning on the feasibility of completing the runtime repository by implementing the missing features.
9.1.2 Parser

Parsing and automagically generation of code that handles much of the QoS management is important to secure the success of CQML.

We have seen that a parser for CQML can be implemented in JavaCC in section 7.2.1. From the evaluation in section 8.2.1 we know that it is a simple parser that can build a parse tree in one sweep, and only looking at the next token.

We have also implemented a simple symbol table that can be used for symbol resolution and type checking.

The implementation of the parser shows that there are issues with having a generic interface to a symbol table, and integrate it with an existing object language. It is not obvious that integrating CQML with an object language, and using the symbol table of an existing parser for this language, is the best solution.

The test case in section 7.3 shows that code generated by the parser can be used to create QoS specifications in the repository.

What we have not shown is the generation of code to be used to negotiate and monitor QoS at runtime, but we have shared some thoughts on the subject, and found it to be a hard but feasible task.

We conclude that CQML is easy to parse to check for syntactically correctness, and that it can be used to generate code that plays well with the repository. CQML specifications are well suited to being transferred to and used in a runtime system.

9.2 Future work

As already noted, this work is not a complete implementation of neither the runtime representation of CQML or the CQML parser. In this section we will look at some of the challenges that should be looked further into.

9.2.1 Improvements to the repository

The repository and parser has been implemented with the intent of showing the suitability of using CQML in a running system. There has been taken shortcuts, and some simplifications has been made.

The implementation of the conformance check in qos_statement needs some smarter logic. As implemented, it checks statistical aspects on a one to one basis. The check should be coded to take into account that a constraint on one statistical aspect can implicitly constrain another aspect. For instance, percentile 90 > 5 implies percentile 80 > 5.

The implementation should also be revisited with optimization in mind.
9.2.2 Unimplemented features

There are some minor and some major features that remain to be implemented.

The qos_characteristic lacks support for units. This should not be a big task to implement.

A major task is to represent semantics for qos_characteristics at runtime, and use it for qos_characteristic comparison. The semantics should also be used for generating code for negotiation and monitoring.

Related to the representation of semantics is the support for parameters on qos_characteristic and qos_statements. Another use for the parameters is to allow qos_statements to span more than one interface entity, as it is limited to in this implementation.

9.2.3 Parser

The parser should be extended so that it supports using the repository during parsing.

It should have support for inserting the QoS specifications directly into the repository instead of having to generate code for inserting the specifications.

The parser should also look up symbols in the repository, if it cannot find them in the file being parsed.

Related to this, is the experiment on not using a symbol table, but rather try to insert the specifications into the repository as they are being parsed. This to get rid of the troublesome symbol table, and leave the symbol lookup and type checking to the repository, which already does this anyways.

9.2.4 Tools

A task more related to CQML itself than to the repository is to define a UML representation of CQML, and to integrate it in a graphical modelling tool.

The next step would then be to integrate the modelling tool with the repository. Allowing the tool to use objects from the repository, and insert new ones.

9.2.5 Benchmarks

When the runtime repository and generated code is mature enough to be used in a real production system, we can run some benchmarks to verify that it can be used in a real time system.

We must first define the benchmarks. What tests are important to run to verify if a system is good enough to be used in a system with real time requirements? Second, we must run the benchmark on well defined hardware.
Benchmarking should also be done in a framework that has better support for QoS and streams than CORBA.
Appendix A

IDL

module CQML {

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;

// the big mother object, currently just plain empty
interface QRObj {
    readonly attribute DefinitionKind defKind;
};

typedef sequence<QRObj> QRObjSeq;

interface Container;
interface Contained : QRObj {
    readonly attribute Identifier id;
    Container get_container();
};
typedef sequence<Contained> ContainedSeq;

// container for storing qr things, and looking them up
interface Container : QRObj {
    // make a new qos-object, and add to container
    // guess i’ll need a name-crash exception (todo)
    qos_characteristic create_qos_characteristic( in Identifier id,
                                              in DomainKind dom )
        raises (NameCrash);

c_qos_statement create_qos_statement( in Identifier id )
    raises (NameCrash);

c_qos_category create_qos_category( in Identifier id )

88
raises (NameCrash);

c_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_val;
};

interface qos_characteristic : Contained {

    // i am someone, and this is my name
    //readonly attribute Identifier id;

    // characteristic is a refinement of some other stuff
    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);

// function to test if this characteristic has equal semantics
// to the argument characteristic, trivial? :)  
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 );

};

// i usually live inside qos_statement, and i’m not sure it’s   
// necessary for me to expose my interface   
// nope, dont think so   
//interface qos_constraint {

    // you might wonder, ”am i stronger than my argument?”   
    //    boolean conforms_to(in qos_constraint constr);

//};

struct AspectDescription {
    AspectKind kind;
    AspectParam param;
    relOp op;
    Value val;

};
typedef sequence<AspectDescription> AspectDescriptionSeq;

struct ConstraintDescription {
    qos_characteristic characteristic;
    AspectDescriptionSeq aspects;
};

typedef sequence<ConstraintDescription> ConstraintDescriptionSeq;

interface qos_statement : Contained {
    // i have a name, so there
    //readonly attribute Identifier id;

    // refines this statement
    void refines(in qos_statement base);

    // strenghten the statement with yet another constraint
    // as you might see, it is only for numeric domain
    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 {
    // my name is
    //readonly attribute Identifier id;
};

91
interface qos_reference : Contained {
    Contained get_reference();
};

interface QoSRepository : Container {

};


Appendix B

JavaCC code

```java
options
{
    MULTI=true;
    NODE_DEFAULT_VOID=true;
    VISITOR=true;
}

PARSER_BEGIN(CQMLParser)

package parser;

import datamodel.*;
import log.*;
import java.util.*;

public class CQMLParser implements ErrorConstants {
    static protected SymbolTable symtab;
    static protected ErrorReporter error_reporter;

    public void setErrorReporter(ErrorReporter er){
        error_reporter = er;
    }

    public void setSymbolTable(SymbolTable sb){
        symtab = sb;
    }

    public static void main(String args[]) throws ParseException
    {
        ErrorGuy errorer = new ErrorGuy();
        symtab = new SymbolTable("root", null);
        error_reporter = errorer;
        errorer.debug(true);
        CQMLParser parser = new CQMLParser(System.in);
        ASTStart n = parser.Start();
        System.out.println(errorer.getNumErrors() + " errors, " +
        + errorer.getNumWarnings() + " warnings.");
        //n.dump("\n");
        //symtab.dump("\n");
        //CQMLParserVisitor v = new CQMLSemanticVisitor();
        //n.jjtAccept(v, null);
    }
}
```
PARSER_END(CQMLParser)

SKIP:
{
    " "
    | "\t"
    | "\n"
    | "\r"
}

/* COMMENTS */
/* they should behave pretty much java comments, because that's where i
* grabbed them. */

MORE :
{
    "/" : IN_SINGLE_LINE_COMMENT
    | <"/**" ~["/"]> { input_stream.backup(1); } : IN_FORMAL_COMMENT
    | "/" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
{
    <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
}

<IN_FORMAL_COMMENT>
SPECIAL_TOKEN :
{
    <FORMAL_COMMENT: "*/" > : DEFAULT
}

<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN :
{
    <MULTI_LINE_COMMENT: "*/" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT,IN_FORMAL_COMMENT,IN_MULTI_LINE_COMMENT>
MORE :
{
    < "[] >
}

ASTStart Start() #Start :
{
    jjtThis.setSymtabEntry(symtab);
}
{
    ( definition(symtab, jjtThis) )* <EDF>
    {return jjtThis; }
}
void definition(SymbolTableEntry se, ASTStart node) {
    qos_definition(se, node.container())
    | ODL_definition(se, node)
}

/* qos definitions */
void qos_definition(SymbolTableEntry se, Container node) {
    ASTqos_characteristic qc;
    ASTqos_category cat;
    ASTqos_statement qs;
}

{| q = qos_characteristic_dcl(se)
    { node.addCharacteristic(qc); qc.setContainer(node); }
| c = qos_category_dcl(se)
    { node.addCategory(cat); cat.setContainer(node); }
| s = qos_statement_dcl(se)
    { node.addStatement(qs); qs.setContainer(node); }
    // OCL_operation_spec(se)
}

/* qos characteristic constructs */
/* could of course make this a forward_dcl, or a qos_char_def */
ASTqos_characteristic qos_characteristic_dcl(SymbolTableEntry se) #qos_characteristic:
{| newTable = qos_characteristic_header(se, jjtThis)
    "(" qos_characteristic_body(newTable, jjtThis) ")"
    { return jjtThis; }
}
SymbolTableEntry qos_characteristic_header(SymbolTableEntry se, ASTqos_characteristic node) {
{| name = identifier()
    { node.setName(name);
        newTable = se.addNewTable(name, node);
        if(newTable == null){
            // this identifier has been used already
            line = getToken(0).beginLine;
            error_reporter.error(ERR_NAME_CONFLICT, line, name + " already declared.");
            // making a dummy table to continue semantic analysis
            do{

32

95

name = name + "*%";
newTable = se.addNewTable(name, node);
while(newTable == null);
}
node.setSymtabEntry(newTable);
}
[params = qos_formal_parameters(newTable)
{ node.setParameters(params); } ]
[qos_char_inheritance_spec(newTable, node)]
{ return newTable; }
}

Vector qos_formal_parameters(SymbolTableEntry se) :
{
  ASTqos_formal_parameter param;
  Vector params = new Vector();
}
{
  "( " [ param = qos_formal_parameter(se)
  { params.addElement(param); } 
  " , " param = qos_formal_parameter(se)
  { params.addElement(param); } 
  )* ] "])
  { params.trimToSize();
    return params;
  }
}

// incomplete - must put type in the node
ASTqos_formal_parameter qos_formal_parameter(SymbolTableEntry se)
#qos_formal_parameter :
{
  String name;
  SymbolTableEntry newEntry;
  int linenum = getToken(1).beginLine;
  jjtThis.setLineNum(linenum);
}
{
  name = identifier()
  {
    // if this is parameter to a qos_characteristic and it is a
    // statistical aspect, then error
    if(se.getCreator() instanceof ASTqos_characteristic &
      ASTqos_statistical_aspect.isAspect(name))
      error_reporter.error(ERR_STAT_ASP_NC, linenum, name +
      " is a reserved keyword in qos_characteristic," +
      " cannot be used as parameter name.");
    else {
      newEntry = se.addNewEntry(name, jjtThis);
      if(newEntry == null)
        error_reporter.error(ERR_PARAM_NC, linenum, name +
        " already declared.");
      else{
        jjtThis.setSymtabEntry(newEntry);
        jjtThis.setName(name);
      }
    }
  }" :" qos_type_spec(se)
  { return jjtThis; }
}
void qos_char_inheritance_spec(SymbolTableEntry se, ASTqos_characteristic node)
{
    ASTscoped_name name;
    SymbolTableEntry inhTable;
    IContained sn;
}

":" name = scoped_name(se)
{
    inhTable = name.getSymtabEntry();
    if(inhTable != null)(
        se.addExtension(inhTable);
        sn = name.getReference();
        while(sn instanceof ASTqos_reference)
            sn = ((ASTqos_reference)sn).getReference();
        // check that sn is a qos_characteristic
        if(sn instanceof ASTqos_characteristic)(
            node.setSuper((ASTqos_characteristic)sn);
            // Is the super the node itself?
            if(sn == node)
                error_reporter.error(ERR_SELF_INHERIT, name.getLineNum(),
                    node.getName() + " inherits itself.");
            else error_reporter.error(ERR_NOT_QOS_CHAR, name.getLineNum(),
                name.getImage() + " is not a qos_characteristic.");
        )
    )
}

ASTscoped_name scoped_name(SymbolTableEntry se) #scoped_name :
{
    SymbolTableEntry curEntry = se;
    SymbolTableEntry parent;
    String name;
    StringBuffer image = new StringBuffer();
    int linenum = getToken(1).beginLine;
    jjtThis.setLineNum(linenum);
    Vector path = new Vector();
    Stack st = new Stack();
}

[ "::"
{
    curEntry = se.getRoot();
    image.append("::");
}
] name = identifier()
{
    image.append(name);
    curEntry = curEntry.findInScope(name);
    if(curEntry == null)
        error_reporter.error(ERR_NAME_UNDEF, linenum, name + " not declared.");
    else {
        parent = curEntry.getParent();
        while(parent != null)(
            st.push(parent.getCreator());
            parent = parent.getParent();
        )
}
st.pop();
while(!st.empty()){
    path.addElement(st.pop());
}
path.addElement(curEntry.getCreator());
}
< "::" name = identifier()
{
    image.append("::" + name);
    if(curEntry != null){
        curEntry = curEntry.findInTable(name);
        if(curEntry == null)
            error_reporter.error(ERR_NAME_UNDEF, linenum,
            name + " not declared.");
    } else {
        path.addElement(curEntry.getCreator());
    }
}
)*
{
    jjtThis.setSymtabEntry(curEntry);
    if(curEntry != null){
        if(curEntry.getCreator() instanceof IContained)
            jjtThis.setReference((IContained)curEntry.getCreator());
        else jjtThis.setSimpleRef(curEntry.getCreator());
    }
    jjtThis.setImage(image.toString());
    jjtThis.setPath(path);
    return jjtThis;
}

// incomplete - doesnt work, never choose door 2
void qos_characteristic_body(SymbolTableEntry se, ASTqos_characteristic node) :
{
    Vector aspects;
}
{
    ([qos_domain_def(node)]
    aspects = qos_statistical_aspects(se)
    { node.setStatisticalAspects(aspects); } [qos_semantics()]
    [qos_invariant()])
    // | qos_characteristic_list() //virker ikke helt :)
}
void qos_domain_def(ASTqos_characteristic node) :
{
    qos_domain n;
    ASTqos_unit unit;
}
{
    "domain" "::"
    ( LOOKAHEAD(2)
        n = qos_numeric_domain()
    | LOOKAHEAD(2) n = qosEnumerated_domain()
    | n = qos_set_domain()
    )
    node.setDomain(n); 
    [unit = qos_unit() { node.setUnit(unit); } ]}
";"
ASTqos_numeric_domain qos_numeric_domain() #qos_numeric_domain : {}
{  
  ("increasing"  
  { jjtThis.setIncreasing(true);  })  
  "numeric"  
  { return jjtThis;  }
}

ASTqosEnumeratedDomain qosEnumeratedDomain() #qosEnumeratedDomain : 
{  
  ASTQosElementSet elems;  
  ASTQosOrder order;
}
{  
  {("enum"  
    elems = qos_element_set()  
    { jjtThis.setOrdered(false);  
      jjtThis.setElements(elems);  
    }  
  )  
  | (  
    "increasing" { jjtThis.setIncreasing(true); }  
    "decreasing" { jjtThis.setIncreasing(false); }  
  )  
  "enum"  
  elems = qos_element_set()  
  "with" order = qos_order(elems)  
  { jjtThis.setElements(elems);  
    jjtThis.setOrdered(true);  
    jjtThis.setOrder(order);  
  }  
  { return jjtThis;  }
}

// incomplete
ASTqosSetDomain qosSetDomain() #qos_set_domain : 
{  
  ASTQosElementSet elems;  
  ASTQosOrder order;  
  jjtThis.hasRelSem(false);  
  jjtThis.setOrdered(false);
}
{  
  {(  
    "increasing" { jjtThis.setIncreasing(true); }  
    | "decreasing" { jjtThis.setIncreasing(false); }  
  )?  
  "set"  
  elems = qos_element_set()  
  { jjtThis.setElements(elems);  
    jjtThis.setOrdered(true);  
    jjtThis.setOrder(order);  
  }  
}
"with" order = qos_order(elems)
{
    jjtThis.setOrdered(true);
    jjtThis.setOrder(order);
}

} { return jjtThis; }

ASTqos_element_set qos_element_set() #qos_element_set:
{
    String name;
}
{
"{"
    name = identifier() { jjtThis.addElement(name); }
    ( "," name = identifier() { jjtThis.addElement(name); } )*?
"
}"
{ return jjtThis; }

ASTqos_order qos_order(ASTqos_element_set elems) #qos_order :
{
    ASTelement_order eord;
}
"order" "{"
    eord = element_order(elems) { jjtThis.addOrder(eord); }
    ( "," eord = element_order(elems) { jjtThis.addOrder(eord); } )*?
"
}"
{ return jjtThis; }

ASTelement_order element_order(ASTqos_element_set elems) #element_order :
{
    String name;
    int linenum = getToken(1).beginLine;
    jjtThis.setLineNum(linenum);
}

{ name = identifier()
{ if(!elems.contains(name)){
    error_reporter.error(ERR_ELEM_UNDEF, linenum, "element " + name + " has not been defined");
}
    jjtThis.setLow(name);
}
"<",
{ name = identifier()
{ if(!elems.contains(name)){
    error_reporter.error(ERR_ELEM_UNDEF, linenum, "element " + name + " has not been defined");
}

jjtThis.setHigh(name);
{ return jjtThis; }
}

// incomplete
ASTqos_unit qos_unit() #qos_unit : {}
{ identifier() ("*"|"/") identifier()*
{ return jjtThis; }
}

Vector qos_statistical_aspects(SymbolTableEntry se) :
{ ASTqos_statistical_aspect aspect;
Vector aspects = new Vector();

{ aspect = qos_statistical_aspect(se) ";";
{ if(aspect != null) aspects.addElement(aspect); }
}

{ if(aspects.isEmpty()) return null;
aspects.trimToSize();
return aspects; }
}

ASTqos_statistical_aspect qos_statistical_aspect(SymbolTableEntry se) :
{ ASTqos_statistical_aspect aspect;
}

{ LOOKAHEAD(2)
aspect = qos_parameterized_aspect(se)
| aspect = qos_simple_aspect(se) }
{ return aspect; }
}

ASTqos_simple_aspect qos_simple_aspect(SymbolTableEntry se)
#qos_simple_aspect :
{ String name;
SymbolTableEntry newEntry;
int linenum = getToken(1).beginLine;
jjtThis.setLineNum(linenum);

{ name = identifier()

//this is not a valid statistical aspect
if(!ASTqos_statistical_aspect.isAspect(name))
   error_reporter.error(ERR_NOT_STAT_ASP, linenum, name + 
   " is not a statistical aspect.");
// it is not a simple aspect
else if(!ASTqos_simple_aspect.isSimpleAspect(name))
   error_reporter.error(ERR_NOT_SIMPLE_ASP, linenum, 
   "statistical aspect " + name + 
   " requires a value or range parameter.");
newEntry = se.addNewEntry(name, jjtThis);
njjtThis.setSymtabEntry(newEntry);
if(newEntry != null){
}
j�This.setName(name);
    return j�This;

} // incomplete - what to do when aspect has been declared here, or
// in super characteristic
error_reporter.warning(WRN_DUP_DECL, linenum, name + 
" is already declared.");
    return null;
}

ASTqos_parameterized_aspect qos_parameterized_aspect(SymbolTableEntry se)

    #qos_parameterized_aspect :
    {
        String name;
        ASTqos_range range;
        SymbolTableEntry newEntry;
        int linenum = getToken(1).beginLine;
        j�This.setLineNum(linenum);
    }

        name = identifier()
    // incomplete only single value, not a range
    range = qos_range()
    {
        //is this really a statistical aspect
        if(!ASTqos_statistical_aspect.isAspect(name))
            error_reporter.error(ERR_NOT_STAT_ASP, linenum, name + 
" is not a statistical aspect.");
        //but is it a parameterized aspect
        else if(!ASTqos_parameterized_aspect.isParameterizedAspect(name))
            error_reporter.error(ERR_NOT_PARAM_ASP, linenum, 
"statistical aspect " + name + 
" should not have any parameters attached to it.");
        newEntry = se.addNewEntry(name, j�This);
        j�This.setSymtabEntry(newEntry);
        if(newEntry != null){
            j�This.setName(name);
            j�This.addRange(range);
            return j�This;
        }
    // this kind of aspect has been declared before, we only add a range
    // to the existing one
        newEntry = se.findInTable(name);
        ASTqos_parameterized_aspect aspect = 
            (ASTqos_parameterized_aspect)newEntry.getCreator();
        aspect.addRange(range);
        return null;
    }
}

// incomplete - must fix to include ranges.
ASTqos_range qos_range() #qos_range :

    {
        int num;
    }

        num = integer_literal()
        {
            j�This.setNum(num);
            return j�This;
void qos_semantics() #qos_semantics : {}
{
    "semantics" ":" OCL_expression() ";"
}

void qos_invariant() #qos_invariant : {}
{
    "invariant" ":" OCL_expression() ";"
}

ASTqos_category qos_category_dcl(SymbolTableEntry se) #qos_category :
{
    SymbolTableEntry newTable;
}
{
    newTable = qos_category_header(se, jjtThis)
    "{" qos_category_body(newTable, jjtThis) "}"
    return jjtThis; }
}

SymbolTableEntry qos_category_header(SymbolTableEntry se, ASTqos_category node) :
{
    String name;
    SymbolTableEntry newTable;
    int linenum = getToken(1).beginLine;
    node.setLineNum(linenum);
}
{
    "qos_category" name = identifier()
    }
    node.setName(name);
    newTable = se.addNewTable(name, node);
    if(newTable == null){
        // agh, the name has been used already
        linenum = getToken(0).beginLine;
        error_reporter.error(ERR_NAME_CONFLICT, linenum, 
        name + " already declared.");
        // making a dummy table to continue semantic analysis
        do{
            name = name + "%";
            newTable = se.addNewTable(name, node);
        } while(newTable == null);
    }
    node.setSymtabEntry(newTable);
    return newTable;
}
}

void qos_category_body(SymbolTableEntry se, ASTqos_category node) :
{
    ASTqos_reference qref;
    
    (qos_definition(se, node)
    | (qref = qos_reference(se) ";"
    { node.addReference(qref); qref.setContainer(node); })

103
```
ASTqos_reference qos_reference(SymbolTableEntry se)
#qos_reference :
{   
ASTscoped_name name;
IContained sn;
}
{   
name = scoped_name(se)
{
    sn = name.getReference();
    if(sn == null) return null; // should never happen
    if((sn instanceof ASTqos_characteristic) ||
        (sn instanceof ASTqos_category) ||
        (sn instanceof ASTqos_statement) ||
        (sn instanceof ASTqos_reference))
    {
        jjtThis.setReference((IContained)sn);
        se.addNewEntry(sn.getName(), jjtThis);
    }
    else {
        inError = true;
        error_reporter.error(ERR_NOT_QOS_CHAR, name.getLineNum(),
            name.getImage() + " is not a qos_characteristic.");
        return null;
    }
}
//return (ASTqos_characteristic)sn;
return jjtThis;
}

ASTqos_statement qos_statement_dcl(SymbolTableEntry se)
#qos_statement :
{   
SymbolTableEntry newTable;
}
{   
newTable = qos_statement_header(se, jjtThis)
"{" qos_statement_body(newTable, jjtThis) "}"
{ return jjtThis;
}

SymbolTableEntry qos_statement_header(SymbolTableEntry se,
    ASTqos_statement node) :
{   
String name;
SymbolTableEntry newTable;
Vector params;
int linenum = getToken(1).beginLine;
node.setLineNum(linenum);
}
{   
"qos" name = identifier()
{
    node.setName(name);
    newTable = se.addNewTable(name, node);
    if(newTable == null){
        // awgh, the name has been used already
        linenum = getToken(0).beginLine;
        error_reporter.error(ERR_NAME_CONFLICT, linenum,
```
name + " already declared.");
// making a dummy table to continue semantic analysis
do{
    name = name + "*%";
    newTable = se.addNewTable(name, node);
    } while(newTable == null);
}
node.setSymtabEntry(newTable);
}

[params = qos_formal_parameters(newTable)
{ node.setParameter(params); } ]
[qos_stmt_inheritance_spec(newTable, node)]
{ return newTable; }
}
void qos_stmt_inheritance_spec(SymbolTableEntry se, ASTqos_statement node) : 
{
    ASTscoped_name name;
    SymbolTableEntry inhTable;
    IContained sn;
}
{"." name = scoped_name(se)
 { inhTable = name.getSymtabEntry();
   if(inhTable != null){
      se.addExtension(inhTable);
      sn = name.getReference();
      while(sn instanceof ASTqos_reference)
        sn = ((ASTqos_reference)sn).getReference();
      // check that sn is a qos_statement
      if(sn instanceof ASTqos_statement){
        node.setSuper((ASTqos_statement)sn);
        // is the super the node itself?
        if(sn == node)
          error_reporter.error(ERR_SELF_INHERIT, name.getLineNum(),
          name.getImage() + " inherits itself.");
        else error_reporter.error(ERR_NOT_QOS_STMT, name.getLineNum(),
          name.getImage() + " is not a qos statement.");}
    }
}
}
void qos_statement_body(SymbolTableEntry se, ASTqos_statement node) :
{
    Vector constraints = new Vector();
    ASTqos_constraint constraint;
}
{ constraint = qos_constraint(se) ";"
 { constraints.addElement(constraint); }
}*
{ constraints.trimToSize();
   node.setConstraints(constraints); }
}
ASTqos_constraint qos_constraint(SymbolTableEntry se) #qos_constraint :
{
    ASTscoped_name name;

105
IContained sn;
ASTqos_characteristic qc;
ASTqos_constr_aspect asp;
ASTqos_constr_value val;
ASTOCL_relationalOperator op;
SymbolTableEntry st = null;

int linenum = getToken(0).beginLine;
jjtThis.setLineNum(linenum);
}
{
  name = scoped_name(se)
  {
    sn = name.getReference();
    while(sn instanceof ASTqos_reference)
      sn = ((ASTqos_reference)sn).getReference();
    if(sn instanceof ASTqos_characteristic){
      qc = (ASTqos_characteristic)sn;
      jjtThis.setCharacteristic(qc);
      st = qc.getSymtabEntry();
    }
    else error_reporter.error(ERR_NOT_QOS_CHAR, name.getLineNum(),
      name.getImage() + " is not a qos_characteristic.");
  }
  [ "(" scoped_name(se) ")" ]
  [ "." asp = qos_constr_aspect(st) ]
  { jjtThis.setStatisticalAspect(asp); }
}

op = OCL_relationalOperator()
val = qos_constr_value()
{ jjtThis.setRelOp(op);
  jjtThis.setValue(val);
  return jjtThis;
}

ASTqos_constr_aspect qos_constr_aspect(SymbolTableEntry se)
#qos_constr_aspect :
{
  int linenum = getToken(1).beginLine;
  jjtThis.setLineNum(linenum);
}
{
  ( LOOKAHEAD(2)
    qos_constr_param_aspect(se, jjtThis)
    | qos_constr_simple_aspect(se, jjtThis)
  )
  { return jjtThis; }
}

void qos_constr_simple_aspect(SymbolTableEntry se, ASTqos_constr_aspect node) :
{
  String name;
  int linenum = getToken(1).beginLine;
}
{
  name = identifier()
  {
    //is this really a statistical aspect
    if(!ASTqos_statistical_aspect.isAspect(name))
      

error_reporter.error(ERR_NOT_STAT_ASP, linenum, name + " is not a statistical aspect.");
//and is it simple
else if(!ASTqos_simple_aspect.isSimpleAspect(name))
    error_reporter.error(ERR_NOT_SIMPLE_ASP, linenum, "statistical aspect " + name + " requires a value or range parameter.");
//and finally, has it been defined in the characteristic
if(se != null){
    se = se.findInTable(name);
    if(se == null)
        error_reporter.error(ERR_ASP_UNDEF, linenum, "statistical aspect " + name + " not defined for qos_characteristic.");
}
node.setName(name);

void qos_constr_param_aspect(SymbolTableEntry se, ASTqos_constr_aspect node) {
    String name;
    ASTqos_parameterized_aspect qa;
    ASTqos_range range;
    int linenum = getToken(1).beginLine;
{
    name = identifier()
    range = qos_range()
{
    //is this really a statistical aspect
    if(!ASTqos_statistical_aspect.isAspect(name))
        error_reporter.error(ERR_NOT_STAT_ASP, linenum, name + " is not a statistical aspect.");
    //and is it parameterized
    else if(!ASTqos_parameterized_aspect.isParameterizedAspect(name))
        error_reporter.error(ERR_NOT_SIMPLE_ASP, linenum, "statistical aspect " + name + " requires a value or range parameter.");
    //and finally, has it been defined in the characteristic
    if(se != null){
        se = se.findInTable(name);
        if(se == null)
            error_reporter.error(ERR_ASP_UNDEF, linenum, "statistical aspect " + name + " not defined for qos_characteristic.");
        qa = (ASTqos_parameterized_aspect)se.getCreator();
        if(!qa.hasRange(range))
            error_reporter.error(ERR_ASP_UNDEF, linenum, "statistical aspect " + name + " not defined for range in" + " qos_characteristic.");
    }
}
node.setName(name);
node.setRange(range);
//incomplete - only integer values
ASTqos_constr_value qos_constr_value() #qos_constr_value :
{  
   int num;
   String elem;
   ASTqos_element_set vset;
}
{  
   (  
      ( num = integer_literal() { jjtThis.setNum(num); } ) 
   | ( elem = identifier() { jjtThis.setElement(elem); } ) 
   | ( vset = qos_element_set() { jjtThis.setSet(vset); } ) 
   )
   { return jjtThis; }
}

ASTqos_spec CQML_qos_spec(SymbolTableEntry se) #qos_spec :
{  
   ASTqos_spec_expr expr;
}
{  
   (  
      [ expr = qos_requires_spec(se)  
         { jjtThis.setRequires(expr); } ] 
   | [ expr = qos_provides_spec(se)  
         { jjtThis.setProvides(expr); } ] 
   )
   { return jjtThis; }
}

ASTqos_spec_expr qos_requires_spec(SymbolTableEntry se) :
{  
   ASTqos_spec_expr expr;
}
{  
   "requires" expr = qos_spec_expr(se) ";"  
   { return expr; }
}

ASTqos_spec_expr qos_provides_spec(SymbolTableEntry se) :
{  
   ASTqos_spec_expr expr;
}
{  
   "provides" expr = qos_spec_expr(se) ";"  
   { return expr; }
}

ASTqos_spec_expr qos_spec_expr(SymbolTableEntry se) :
{  
   ASTqos_spec_expr expr;
}
{  
   expr = qos_or_expr(se)  
   { return expr; }
}

ASTqos_spec_expr qos_or_expr(SymbolTableEntry se) :
{
ASTqos_spec_expr expr;
{
  ((
    expr = qos_and_expr(se)
    { jjtThis.addExpr(expr); }
    "or" expr = qos_and_expr(se)
    { jjtThis.addExpr(expr); }
  )* 
  )
  if(jjt.tree.nodeCreated()) return jjtThis;
  else return expr;
})
#qos_or_expr(>1)
}

ASTqos_spec_expr qos_and_expr(SymbolTableEntry se) :
{
  ASTqos_spec_expr expr;
  {
  ((
    expr = qosPrimary_expr(se)
    { jjtThis.addExpr(expr); }
    "and" expr = qosPrimary_expr(se)
    { jjtThis.addExpr(expr); }
  )* 
  )
  if(jjt.tree.nodeCreated()) return jjtThis;
  else return expr;
})
#qos_and_expr(>1)
}

ASTqos_spec_expr qosPrimary_expr(SymbolTableEntry se) :
{
  ASTqos_spec_expr expr;
  {
    ("(" expr = qosSpec_expr(se) ")"
    | expr = qosStmt_expr(se)
    )
    { return expr; }
  }
}

ASTqos_spec_expr qosStmt_expr(SymbolTableEntry se) 
#qosStmt_expr :
{
  ASTscoped_name name;
  SymbolTableEntry inhTable;
  IContained sn;
}
{
  name = scoped_name(se)
  {
    inhTable = name.getSymtabEntry();
    if(inhTable != null){
      se.addExtension(inhTable);
      sn = name.getReference();
    }
  }
}
while(sn instanceof ASTqos_reference)
    sn = ((ASTqos_reference)sn).getReference();
// check that sn is a qos_statement
if(sn instanceof ASTqos_statement){
    jjtThis.setStatement((ASTqos_statement)sn);
} else error_reporter.error(ERR_NOT_QOS_STMT, name.getLineNum(),
    name.getImage() + " is not a qos statement.");
}

"(" name = scoped_name(se) "")"
{
    jjtThis.setInterfaceEntity(name);
}
{return jjtThis;
}

void qos_type_spec(SymbolTableEntry se) : {}
{
    qos_meta_type_spec()
    | ODL_param_type_spec(se)
}

void qos_meta_type_spec() : {}
{
    "Operation"
    | "Stream"
}

int integer_literal() :
{
    Token t;
    { t = <DECIMALINT> 
        { return Integer.parseInt(t.image); } }
}

String identifier() :
{
    Token t;
    { t = <ID> 
        { return t.image; } }
}

void OCL_operation_spec(SymbolTableEntry se) #OCL_operation_spec :
{
    SymbolTableEntry newTable;
}
{ 
    newTable = OCL_operation_sign(se, jjtThis)
    [ OCL_pre_cond() ]
    [ OCL_post_cond() ]
SymbolTableEntry OCL_operation_sign(SymbolTableEntry se, SimpleNode parent)

#OCL_operation_sign :
{
    String name;
    Vector path;
    SymbolTableEntry newTable;
}
{
    path = OCL_pathName()
    {
        newTable = se;
        for(int i=0; i<(path.size()-1); i++){
            name = (String)path.elementAt(i);
            if(newTable.inTable(name)){
                newTable = newTable.getEntry(name);
            } else {
                newTable = newTable.addNewTable(name, null);
            }
        }
        name = (String)path.lastElement();
        newTable = newTable.addNewTable(name, parent);
    }
    qos_formal_parameters(newTable) "::" OCL_return_type(se)
    {
        return newTable; }
}

void OCL_return_type(SymbolTableEntry se) #OCL_return_type : {}
{
    qos_type_spec(se)
}

void OCL_pre_cond() #OCL_pre_cond : {}
{
    "pre " :: OCL_expression()
}

void OCL_post_cond() #OCL_post_cond : {}
{
    "post " :: OCL_expression()
}

/* taken from "UML Object Constraint Language Specification" * 
* or ad/97-08-08 if you like */

void OCL_expression() #OCL_expression : {}
{
    OCL_logicalExpression()
}

void OCL_ifExpression() #OCL_ifExpression : {}
{
    "if" OCL_expression()
    "then" OCL_expression()
    "else" OCL_expression()
    "endif"
void OCL_logicalExpression() : {}
{
    (OCL_relationalExpression()
     (OCL_logicalOperator() OCL_relationalExpression())*) #OCL_logicalExpression(>1)
}

void OCL_relationalExpression() : {}
{
    (OCL_additiveExpression()
     (OCL_relationalOperator() OCL_additiveExpression())?)
    #OCL_relationalExpression(>1)
}

void OCL_additiveExpression() : {}
{
    (OCL_multiplicativeExpression()
     (OCL_addOperator() OCL_multiplicativeExpression()))
    #OCL_additiveExpression(>1)
}

void OCL_multiplicativeExpression() : {}
{
    (OCL_unaryExpression()
     (OCL_multiplyOperator() OCL_unaryExpression()))
    #OCL_multiplicativeExpression(>1)
}

void OCL_unaryExpression() : {}
{
    (OCL_unaryOperator() OCL_postfixExpression())
    #OCL_unaryExpression(>1)
}

void OCL_postfixExpression() : {}
{
    (OCL_primaryExpression() ("." | "->") OCL_featureCallOrStatAspect())
    #OCL_postfixExpression(>1)
}

void OCL_primaryExpression() : {}
{
    OCL_literalCollection()
    | OCL_literal()
    | LOOKAHEAD({ ASTqos_statistical_aspect.isAspect(getToken(1).image) })
    OCL_statistical_aspect()
    | (OCL_pathName() OCL_timeExpression())? (OCL_qualifiers())?
    (OCL_featureCallParameters())?
    #OCL_primary_expression(>1)
    | "(" OCL_expression() ")"
    | OCL_ifExpression()
}

void OCL_featureCallOrStatAspect() : {}
{
    LOOKAHEAD({ ASTqos_statistical_aspect.isAspect(getToken(1).image) })
void OCL_statistical_aspect() : {} 
{ 
  identifier() [ integer_literal() ] 
} 

void OCL_featureCallParameters() : {} 
{ 
  "(" (LOOKAHEAD(OCL_declarators()) OCL_declarators())? 
  (OCL_actualParameterList())? ")" 
) #OCL_featureCallParameters(>1) 
} 

void OCL_literal() #OCL_literal : {} 
{ 
  OCL_string() | OCL_number() | "#" OCL_name() 
} 

void OCL_enumerationType() #OCL_enumerationType : {} 
{ 
  "enum" "{" "#" OCL_name() ( "," "#" OCL_name() )* "}" 
} 

void OCL_simpleTypeSpecifier() : {} 
{ 
  OCL_pathTypeName() 
  | OCL_enumerationType() 
} 

void OCL_literalCollection() # OCL_literalCollection : {} 
{ 
  OCL_collectionKind() "{" (OCL_expressionListOrRange())? "}" 
} 

void OCL_expressionListOrRange() : {} 
{ 
  OCL_expression() 
  ( ( "," OCL_expression() )* 
  | ( ".." OCL_expression() )? 
  )? 
} 

void OCL_featureCall() #OCL_featureCall : {} 
{ 
  OCL_pathName() (OCL_timeExpression())? (OCL_qualifiers())? 
  (OCL_featureCallParameters())? 
} 

void OCL_qualifiers() #OCL_qualifiers : {} 
{ 
  "[" OCL_actualParameterList() "]" 
} 

// add construct, so that iterate will work as well 
void OCL_declarators() #OCL_declarators : {} 
{ 
  OCL_declarator() ( ";" OCL_declarator() )* "]" 
}
void OCL_declarator() #OCL_declarator : {}
{
    OCL_name() ("," OCL_name() )*
    (":" OCL_simpleTypeSpecifier() )?
    ("=" OCL_expression() )?
}

void OCL_pathTypeName() #OCL_pathTypeName : {}
{
    OCL_typeName() ("::" OCL_typeName() )*
}

/* changed: replaced (OCL_name() | OCL_typeNAME()) with identifier() */
Vector OCL_pathName() #OCL_pathName :
{
    String name;
    Vector path = new Vector();
    
    ( name = identifier() ) { path.addElement(name); }
    ("::" ( name = identifier() ) { path.addElement(name); })*
    { return path; }
}

void OCL_timeExpression() #OCL_timeExpression : {}
{
    "@" OCL_name()
}

void OCL_actualParameterList() #OCL_actualParameterList : {}
{
    OCL_expression() (""," OCL_expression() )*
}

void OCL_logicalOperator() #OCL_logicalOperator : {}
{
    "and" | "or" | "xor" | "implies"
}

void OCL_collectionKind() #OCL_collectionKind : {}
{
    "Set" | "Bag" | "Sequence" | "Collection"
}

ASTDCL_relationalOperator OCL_relationalOperator()
#OCL_relationalOperator : {}
{
    {
        ("=" { jjtThis.setOp(ASTDCL_relationalOperator.eq); })
        (">" { jjtThis.setOp(ASTDCL_relationalOperator.gt); })
        ("<" { jjtThis.setOp(ASTDCL_relationalOperator.lt); })
        (">=" { jjtThis.setOp(ASTDCL_relationalOperator.ge); })
        ("<=" { jjtThis.setOp(ASTDCL_relationalOperator.le); })
        ("<>" { jjtThis.setOp(ASTDCL_relationalOperator.ne); })
    }
    { return jjtThis; }
}

void OCL_addOperator() #OCL_addOperator : {}
{
    "+" | "-
}
void OCL_multiplyOperator() #OCL_multiplyOperator : {}
{
    "*" | "/"
}

void OCL_unaryOperator() #OCL_unaryOperator : {}
{
    "-" | "not"
}

void OCL_typeName() #OCL_typeName : {}
{
    <ID>
}

void OCL_name() #OCL_name : {}
{
    <ID>
}

void OCL_number() #OCL_number : {}
{
    <DECIMALINT>
}

void OCL_string() #OCL_string : {}
{
    <OCL_STRING>
}

---< SNIP! cuts away the ODL part of the parser code >---

TOKEN :
{
    < ID : ["a"-"z","A"-"Z","_"] (["a"-"z","A"-"Z","0"-"9","_"])*> >
    | < DECIMALINT : ["1"-"9"] (["0"-"9"])*> (\u",","1","L"))? >
    | < OCL_STRING : "" (\"[\"\",\\"\",\"n","\t"]\")
        | (\"[\"n","t","b","r","f","\",","\",","\",","\",","\",","\",","\",","\",","\",""]
        | ["0"-"7"] (\"[\"0"-"7"]\")?
        | ["0"-"3"] ["0"-"7"] ["0"-"7"]
    )
    | )*> >
}
Appendix C

QoS runtime library

In this appendix, we present the QoS framework that are used as a basis for the code generated by the parser. These classes are wrappings for the CORBA classes.

```cpp
#ifndef __QoSLib_h
#define __QoSLib_h
#include "QosRuntime.h"
//class container_base;
//class qr_base;

// a proxy for the proxy, so to speak
class qrobj_base :
virtual public QRObject_skel {
protected:
    QRObject_ptr obj;
    //    container_base *cont;

public:
qrobj_base(){
    obj = NULL;
    //    cont = NULL;
}
void ref(QRObject_ptr op){
    obj = op;
}
QRObject_ptr ref(){
    if(obj == NULL) create();
    return obj;
}
virtual void create() {}
    // void container(container_base &cp);
    // container_base &container();
```
// virtual qr_base *repository();

DefinitionKind defKind();
};

class contained_base :
virtual public qrobj_base,
virtual public Contained_skel
{
  public:
    void ref(Contained_ptr cp){
      qrobj_base::ref(cp);
    }

    Contained_ptr ref(){
      return Contained::narrow(qrobj_base::ref());
    }

    char *id();

    Container_ptr get_container();
};
class qc_base :
virtual public contained_base,
virtual public qos_characteristic_skel
{
  public:
    void ref(qos_characteristic_ptr qcp){
      contained_base::ref(qcp);
    }

    qos_characteristic_ptr ref(){
      return qos_characteristic::narrow(contained_base::ref());
    }

    void refines( qos_characteristic_ptr base);

    void domain_values( const IdentifierSeq& vals );

    void domain_order( const ValueOrderSeq& rels );

    void simple_aspect( AspectKind ak );

    void param_aspect( AspectKind ak, const AspectParam& param );

    CORBA::Boolean simple_aspect_defined( AspectKind ak );

    CORBA::Boolean param_aspect_defined( AspectKind ak, const AspectParam& param );

    CORBA::Boolean has_equal_semantics( qos_characteristic_ptr qc );

    CORBA::Boolean value_in_domain( const Value& val );

    CORBA::Short compare( const Value& strong, const Value& weak );
};
class qs_base :
    virtual public contained_base,
    virtual public qos_statement_skel
    {
        public:

            qos_statement_ptr ref(){
                return qos_statement::_narrow(contained_base::ref());
            }

            void ref(qos_statement_ptr qsp){
                contained_base::ref(qsp);
            }

            void refines(qos_statement_ptr base);

            void add_constraint(qos_characteristic_ptr qc,
                const AspectDescription &aspect);

            CORBA::Boolean conforms_to(qos_statement_ptr qs);

            CORBA::Boolean conforms_to(qs_base &qs);

            ConstraintDescriptionSeq *constraints();
    }

class container_base :
    virtual public qrobj_base,
    virtual public Container_skel
    {
        public:

            void ref(Container_ptr cp){
                qrobj_base::ref(cp);
            }

            Container_ptr ref(){
                return Container::_narrow(qrobj_base::ref());
            }

            qos_characteristic_ptr create_qos_characteristic(const char *id,
                DomainKind dom);

            qos_statement_ptr create_qos_statement(const char *id);

            qos_category_ptr create_qos_category(const char *id);

            qos_reference_ptr create_qos_reference(Contained_ptr qo);

            Contained_ptr find_name(const char *name);

            ContainedSeq *list();

            void remove(Contained_ptr qo);
    }
class qcat_base :
virtual public contained_base,
virtual public container_base,
virtual public qos_category_skel
{
public:

    void ref(qos_category_ptr qcp){
        contained_base::ref(qcp);
    }

    qos_category_ptr ref(){
        return qos_category::narrow(contained_base::ref());
    }
};

class qref_base :
virtual public contained_base,
virtual public qos_reference_skel
{
public:

    void ref(qos_reference_ptr qrp){
        contained_base::ref(qrp);
    }

    qos_reference_ptr ref(){
        return qos_reference::narrow(contained_base::ref());
    }

    Contained_ptr get_reference();
};

class qr_base :
virtual public container_base,
virtual public QoSRepository_skel
{
public:

    void ref(QoSRepository_ptr qrp){
        container_base::ref(qrp);
    }

    QoSRepository_ptr ref(){
        return QoSRepository::narrow(container_base::ref());
    }

    // virtual qr_base *repository();
};

#endif
Bibliography


