Using UML to specify QoS constraints in ODP

Behzad Bordbar, John Derrick, Gill Waters
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Email: B.Bordbar@ukc.ac.uk.)

January 8, 2002

Abstract

This paper is concerned with Quality of Service (QoS) specification in distributed system design. The specification and implementation of QoS is increasingly important in distributed systems due to the need to address questions of performance, particularly for systems involving multimedia. To ensure correct implementation of QoS requirements, statements of QoS need to be introduced early in the design process, and in terms of design we consider the use of the Unified Modelling Language (UML), which has quickly become the de facto standard for object-based designs.

The framework we use for distributed system construction is that provided by the Open Distributed Processing reference model, and we focus in particular on its computational viewpoint. The aim of this paper is to construct a UML model of the computational viewpoint focusing on the description of QoS within that viewpoint. To specify the QoS aspects of computational objects in the UML model, we use a real-time logic called QL. In order to express further constraints on the UML model of the computational viewpoint, we use the Object Constraint Language (OCL) to express invariants that each instance of our model must satisfy. The purpose of our UML model of the computational viewpoint is to act as a template via which specific distributed system designs can be constructed and we illustrate this with the specification of a lip synchronisation mechanism.

Keywords: Open Distributed Processing; Computational Viewpoint; Quality of Service; Specification; UML; Object Constraint Language.

1 Introduction

The design of distributed systems is non-trivial and the advent of multimedia systems has increased their complexity. One aspect of this complexity is the central role that questions of performance play in the specification and implementation of such systems. However, traditional systems engineering methodologies do not focus on the development of distributed systems, particularly with respect to issues of performance and their associated Quality of Service (QoS) constraints.

Therefore, because modern distributed systems are object-based, there has been considerable interest in the use of the Unified Modelling Language (UML) [36] for the design of complex distributed systems. Indeed, UML has quickly become a de facto standard for visualising, specifying, designing, and documenting object-oriented systems, and has emerged as the standard object-oriented analysis and design notation. However, because it attempts to provide notations

for most aspects of object-oriented design, users can select from a rich variety of design diagrams in UML and there are few guidelines on how precisely the design is defined.

The purpose of this paper is to present one approach to the specification of QoS requirements using UML. The framework in which we present these ideas is that offered by the Open Distributed Processing (ODP) architecture. Open Distributed Processing [16] is a joint ITU/ISO standardisation framework for constructing distributed systems in a multi-vendor environment. Significant features of ODP include *object based* specification and programming, use of *transparencies* to hide aspects of distribution and its use of *viewpoints*.

ODP viewpoints are used to help partition the complexity in distributed systems design. Each viewpoint considers the specification of a distributed system from a particular perspective, and of particular relevance to QoS specification is the *computational viewpoint*. This viewpoint is concerned with the algorithms and data flows which provide the distributed system function. It represents the system and its environment in terms of objects which interact by transfer of information via interfaces. This does not necessarily imply that the computational objects will be realized in the eventual system by separate distributed components, but indicates the candidate objects from which components can be chosen.

QoS is central to the ODP architecture, and impacts across all viewpoints. In terms of the computational description QoS is specified in a contractual approach which describes both the QoS required by a computational object and the QoS provided by that object. This approach is taken so that the overall QoS requirement can be broken down into appropriate dependencies between groups of computational objects.

In order to be able to clearly express these dependencies, QoS specification needs to be considered along with aspects of the functional description, and here we consider how UML can be used to support this. There has been considerable interest in the use of UML in distributed systems design in general and ODP in particular (see for example [2, 9, 20, 35, 31]). But they do not elaborate on specifying the QoS aspects of the underlying model of the distributed system. To enable design to include appropriate QoS specification, we provide a UML metamodel of the computational viewpoint with QoS annotation given as attributes of the interface to each metaclass. In order to express further constraints on the metaclass, we use OCL [37] expressions in the form of invariants that each instance of our model must satisfy. For each specific domain of application, we shall create an instance of the metamodel of the computational viewpoint, which is a class diagram capturing its static aspects. Such a class diagram acts as a template from which object diagrams of an application can be instantiated. Such object diagrams, which are static models of each application, embody details of the components of the distributed system, their relationship and QoS details associated to each component.

The purpose of such a UML model is to act as a template via which specific distributed system designs can be constructed, and to illustrate this we show how the UML model can be used to produce an object diagram for a particular design (here part of a lip synchronisation mechanism for audio and video channels).

The structure of the paper is as follows. In Section 2 we provide a brief overview of ODP. Section 3 considers the role of QoS and introduces our running example along with the language QL via which we will specify QoS constraints. UML and OCL are discussed in Section 4 and Section 5 describes how the computational viewpoint is modelled in UML. The example is re-visited in Section 5. Section 6 discusses some related work and we conclude the paper in Section 7.

2 Open Distributed Processing

Open Distributed Processing is a joint ITU/ISO standardisation framework for constructing distributed systems in a multi-vendor environment. The Reference Model for Open Distributed Processing (RM-ODP), which has recently progressed to become an international standard [16], describes an architecture for building open distributed systems [23, 28]. Its scope is broad, including support for all types of traditional data processing systems, networked personal computers, real-time systems and multimedia systems.

One of the major problems in open distributed processing is the wide scope and inherent complexity of the domain. This is reflected in the large amount of information required to produce a complete specification of a distributed system. RM-ODP addresses this problem by introducing the concept of *viewpoints* which are used to partition a system specification into a number of partial descriptions, each targeted towards a particular audience. The reference model defines five viewpoints, namely:

- Enterprise viewpoint: The enterprise viewpoint specifies the scope and objectives of the system and considers the role of the distributed system within the context of an enterprise. For example, this viewpoint deals with contracts expressing the obligations of various parties involved in the distributed system.
- Information viewpoint: The information viewpoint is concerned with information modelling. By factoring an information model out of the individual components, it provides a common view which can be referenced by the specifications of information sources and sinks and the information flows between them.
- Computational viewpoint: The computational viewpoint is the main focus for functional design, and considers the logical partitioning of the distributed system into a series of interacting entities often referred to as objects. Therefore, in this viewpoint an application is decomposed into a number of objects each offering one or more interfaces through which they interact.
- Engineering viewpoint: The engineering viewpoint presents the mechanisms which support various actions and interactions in the computational viewpoint specification, by defining a set of abstract concepts required to model communication and system resources.
- Technology viewpoint: The technology viewpoint considers the development of a distributed system from the perspective of the identification, procurement and installation of particular hardware or software technologies. In particular, this viewpoint explains the overall design of the model in terms of realisable technology.

Further information regarding both the reference model and its approach to using viewpoints can be found in [16, 28, 23]. In this paper we are concerned mainly with the expression of QoS in the computational viewpoint, and in the rest of this section we sketch the key elements of the computational viewpoint that we need based upon the reference model [16] and ideas presented by Blair and Stefani [6].

2.1 Objects and Interfaces

Based on an object-oriented (OO) approach, an essential feature of ODP is that all interacting entities are uniformly encapsulated as *objects*. The word "object", in common with other OO methodologies, is also a reserved word in UML. In UML, concepts of the real world are modelled

via classes. Objects are instantiations of classes. To be specific in our usage of the word object in the context of computational viewpoint of ODP, we use the term computational object, or *compobj* for short.

To perform a service in a distributed environment, the computational objects involved need to access one another, through (possibly multiple) *interfaces*. In this model interfaces, which provide abstract views of an object, can be added and removed dynamically. Multiple interfaces partition the external behaviour associated with computational objects into logically distinct categories enabling a computational object to interact with more than one other object. For example in a CCTV system, a camera can have two interfaces, one interface for transferring the video frames to the monitor screen and the other to operate the camera, i.e., switch on/off, zoom, tilt etc. To communicate with other computational objects, an interface is known to its environment by its *interface reference*.

The computational viewpoint contains three types of interfaces: *operational*, *stream* and *signal*. Operational interfaces support invocation of operations on potentially remote computational objects. A classical example of an operational interface is an interface offered by a computational object for a *Remote Procedure Call (RPC)* [11]. Operations are either *interrogations* (two way operations, comprising an invocation, followed by a *termination* carrying results or exceptions), or *announcements*, which are one way operations (only an invocation). To facilitate interactions involving continuous media such as audio and video flow, the reference model introduces the idea of a stream interface. A stream interface represents a continuous flow and is modelled by the type of media which it handles, together with the direction of the flow (producer or consumer) and its name. An example of a stream interface might be a produced audio flow representing the output of a microphone object. A signal is an atomic action resulting in a one-way communication from an *initiating object* to a *responding object*. In an abstract sense, streams and operations can be defined in term of signals. This enables signal interfaces to be used as a basis for explaining multi-part, end-to-end quality of service characteristics and compound bindings between different kinds of interfaces (e.g. stream to operation interface bindings).

2.2 Trading and Binding

In order for interactions between objects to occur, the interfaces of the relevant objects are associated by the formation of a *binding*. These bindings can be implicit or explicit. In an implicit binding the objects involved are simply linked together and there is no term for expressing the binding action. Additional information needed can be supplied in an *explicit binding* in which the binding itself is encapsulated as an object which provides the infrastructure resources supporting the communication [6, 16].

Each distributed environment contains some form of *trading service*, which records the relevant interface references and acts as a broker between computational objects. An oversimplified outline of how computational objects access each other in order to provide a service is as follows. A computational object wishing to offer a service passes its interface to the trading service. When another computational object wishes to use the service, it sends its own interface to the trading service to locate a suitable computational object. The trading service then creates an implicit binding between the interfaces of the two computational objects.

An example of a computational description involving the use of objects, interfaces and bindings is given in Section 3.2 below. This will consider the lip synchronisation of audio and video channels and introduces computational objects for the video camera, microphone and speaker (along with others). An explicit audio stream binding connects the microphone and speaker objects via stream interfaces representing the flow of audio signals from the microphone to the speaker.

3 Quality of Service

Quality of Service is a rather general term used to differentiate between performance aspects and functional aspects of distributed systems. In general, since the functions that computational objects provide are subject to delay, error, etc,..., the service provided by the overall system is affected by the quality of provision of such functions. Such attributes (e.g. delay) are referred to as QoS, and in the context of distributed multimedia applications, we shall be interested in the following categories [6]:

- Timeliness dimension: This category includes issues related to the end-to-end delay of continuous or discrete media. For example, *latency* which is the amount of time required from generation of a media frame to its eventual display, and *jitter* which is the variation in nominal latency suffered by successive frames of the same message, are important QoS properties.
- Volume related dimension: This category covers the throughput of data and includes, for example, the total number of frames delivered per second in a video stream or the number of bytes delivered per second in a discrete interaction.

In order that a distributed system meets its required performance, statements of the desired QoS must be made. There are a number of different types of QoS requirements that can be considered. For example, requirements can either have a precise value or range of values or be probabilistic or stochastic in nature. Here we shall concentrate on the former requirements, e.g. those giving precise bounds on the acceptable latency in a system.

In order to meet such QoS requirements, the specification and design of a distributed system must include appropriate QoS constraints and guarantees. In terms of ODP such QoS specifications are relevant to all viewpoints, but clearly the role of the computational viewpoint is central. The QoS associated with a computational object then consists of two clauses: the QoS *provided* to the environment by the computational object and the QoS *required* by the object from its environment [6]. A computational object guarantees to supply the provided QoS to the environment only if it receives the required QoS from the environment. This approach is known as a contractual approach to QoS. We will model this invariant on any computational object as an implication between the required QoS and the provided QoS.

3.1 Signal interfaces and Reactive objects

In order to model the process of ensuring that the required QoS is met, the RM-ODP introduces the idea of a *signal interface*. Signal interfaces, which represent low level support for implementation of real-time synchronisation, are used within real-time controllers (referred to as the *reactive object* in [6]) which are introduced to ensure QoS constraints can be realised. For example, to achieve lip synchronisation [32] a control mechanism is obligatory in order to tune the received video and audio flows. Such control mechanisms interact with both the audio and video channels in real-time via signal interfaces.

Real-time controllers are modelled as objects called *reactive objects.* A reactive object accepts events from the outside world and reacts instantaneously by emitting signals back. Similar to any other object in the computational model, a reactive object interacts with its environment through interfaces, and because of the nature of a reactive object, all its interfaces are signal interfaces.

As in the computational viewpoint QoS, requirements in a reactive object need to be specified, and such behaviour could, for example, be described using ESTEREL [4] or LOTOS [3, 34]. The details of how such constraints are implemented are expressed within the engineering viewpoint, however, the reactive object in the computational viewpoint demands a level of commitment from engineering layer that the constraints are satisfied.

3.2 Example

As an example, consider the synchronisation of video and audio channels used to achieve lip synchronisation (see Fig. 1)[1].

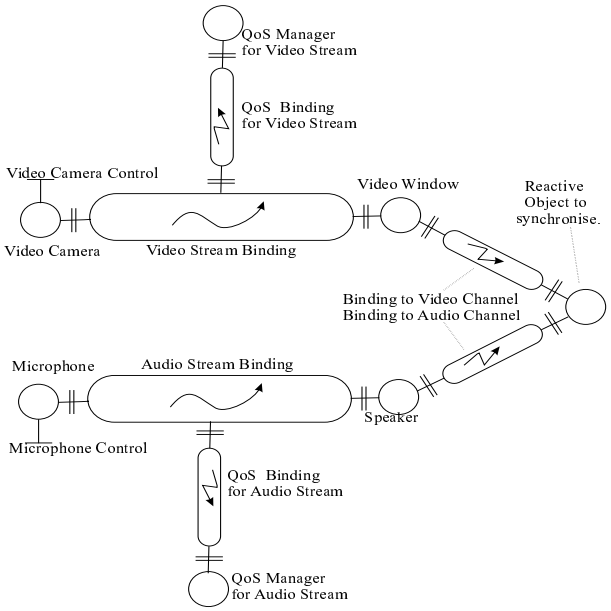

Figure 1: Lip Synchronisation of audio and video stream channels

A specification in the computational viewpoint decomposes the video channel into a video camera and a video window. The video flow stream is carried from video camera into video window via a video stream binding.

In order to achieve lip synchronisation it is necessary to specify appropriate QoS constraints, for example, we require the following of the video flow stream:

- a latency between 40 and 60 ms,
- a maximum jitter of ± 10 ms, and
- a throughput of at least 25 frames per second.

To satisfy the above QoS constraints, a reactive object called *QoS Manager for Video Stream* is provided which interacts with the video stream binding through a signal binding called *QoS Binding for Video Stream.* In case of a failure of the binding, a signal is emitted to indicate

[1] This is based on an example in [6].

violation of the QoS. The audio channel is defined similarly. The QoS for the audio stream is as follows

- a latency between 40 and 60 ms,
- a maximum jitter of ± 200 ms, and
- a throughput of at least 5 packets per second.

To synchronise the audio and video channels a reactive object called *Reactive Object to Synchronise* is created which interacts via signal bindings with the audio and video channels. These bindings are denoted *Binding to Audio Channel* and *Binding to Video Channel* in Fig. 1.

3.3 The language QL

Clearly there needs to be an appropriate way to specify the types of QoS clauses we have just considered in the example above. There are a number of approaches that can be taken, for example [6, 22, 5]. As an example we will consider the use of QL [6], a first order real-time logic based on RTL [18].

QL allows one to specify QoS requirements as predicates involving the timing of permissible events. Events correspond to signals in the computational viewpoint. In particular, each event in the model is written as the concatenation of an interface name, a signal name and a causality. The causality, which is either a consumer or a producer, is denoted by SR (signal received) and SE (signal emission). For example, $VidCamOut.VidOut.SR$ refers to signal received with the name of $VidOut$ in the interface $VidCamOut$.

QoS is intimately tied up with time, and so QL introduces a date function τ to denote the time of occurrence of events. For example, for a natural number n, the expression $\tau(VidCamOut.VidOut.SR, n)$ refers to the date of occurrence of the nth signal $VidCamOut.VidOut.SR$. Predicates can then be constructed using quantifiers $\forall$ and $\exists$ and inequalities such as $\leq, <, \geq$ and $>$, in order to express a large group of QoS clauses. For example, to say that a camera sends out 25 frames in a given second can be written (with time given in ms) as:

$$\forall n, \tau(VidCamOut.VidOut.SE, n + 24) \leq \tau(VidCamOut.VidOut.SE, n) + 1000$$

To enable formal analysis, it is necessary to impose a certain style of QL formulae [6]. The style is as follows:

Rule 1: All basic formulae should have the form

$$\tau(\epsilon_1, n + j_1) + \delta \leq \tau(\epsilon_2, n + j_2)$$

where ϵ_1 and ϵ_2 are concatenations of an interface name, a signal name and a causality, as explained above. n, j_1 and j_2 are non-negative integers variables and δ is a constant integer.

Rule 2: An overall formula can then have the general form:

$$\exists i_1 \ldots \exists i_p \forall n \exists j_1 \ldots \exists j_m \forall n_1 \ldots \forall n_q \beta$$

where β is a conjunction of the basic formulae defined by Rule 1.

Definition 1 *We shall refer to the set of all conjunctions and disjunctions of formulae created via the above Rules as the set of all QL clauses, denoted by QLclause.*

QL clauses can be used to express latency. Latency in a stream interaction, measured in milliseconds, is defined as the time taken from generation of a media frame to its eventual display. For example, assume that ϵ_r and ϵ_s are two events corresponding to sending and receiving of video frames in the stream video channel of Fig. 1. Then to express the latency of 15 ms, we can write

$$\forall n, \quad |\tau(\epsilon_s, n) - \tau(\epsilon_r, n)| \leq 15$$

which says that for all values of n the variation in time of the nth occurrence of ϵ_r and ϵ_s must not exceed 15 ms.

QL clauses can also express jitter and throughput. Jitter, measured in milliseconds is defined as the *variation* in overall nominal latency suffered by individual messages in the same stream. For example assume that ϵ_r and ϵ_S correspond to sending and receiving video frames of a stream video over the channel of Fig. 1. To express the jitter bounded by 20 ms and 30 ms [6], we can write

$$\forall n, \quad 20 \leq |\tau(\epsilon_s, n) - \tau(\epsilon_r, n)| \leq 30.$$

Similarly, for the throughput, we can write

$$\forall n, \quad |\tau(\epsilon_r, n + 5) - \tau(\epsilon_r, n)| \leq 20.$$

to express the requirement that we wish to receive at least 5 frames every 20 ms.

Although QL can specify a large number of QoS requirements, it has several limitations. For example, QL does not support probabilistic and stochastic properties. However, our approach is entirely independent of the choice of the language specifying the QoS requirements. The main reason for choosing QL is its simplicity which helps us to avoid being distracted by the details of specifying a complex language for QoS and enables us to concentrate on explaining the methodology of our modelling approach.

4 Unified Modelling Language and OCL

Object-oriented (OO) concepts are crucial in software design and analysis because they address fundamental issues of adaptation and evolution. Furthermore, most modern distributed systems, and their architectural frameworks, are object-based and call for the use of OO design methodologies for their construction.

The Unified Modelling Language (UML) [8, 36], which combines a number of approaches [7, 17, 27, 19], has become the de facto standard for OO design. UML aims to take the designer through the whole design life cycle, and tries to ensure convergence and clarity in design by prescribing a set of steps starting from the description provided by users or experts down to the final software product. Whether or not this has been successful, UML has gained a certain prominence because it provides engineers with a common format by which designs can be communicated, and in this respect its importance cannot be ignored.

In broad terms, a UML design involves the creation of a series of graphs or charts and the generation of an evolving model of the system. The UML standard recognises nine main diagrams which fall into three categories. *Use case* diagrams deal with the graphical representation of functional requirements of the system in term of interaction between different users and actors.

Static aspect diagrams are used to define the classes of objects, describe their attributes, specify the operations on the classes, describe their relationship or association with other objects, and describe how they are instantiated and interconnected. *Dynamic* aspect diagrams are used to describe the behaviour of the objects in terms of the sequences of operations they perform and the way in which they interact with other connected objects (or components).

This paper is particularly concerned with static aspects of the computational viewpoint of RM-ODP, and in particular we aim to define a template given as a class diagram which will act as a starting point for a distributed system design involving QoS.

Creation of a UML model often starts with recognising different *key agents* of the system, known as *objects* in UML terminology. Considering common features of key agents, objects are extrapolated into collections called *classes.* Classes are depicted by rectangular boxes consisting of two or three segments. The top segment holds the name of the class; the middle segment accommodates a list of *attributes* of the class and the bottom segment contains a list of *methods* (operations) related to the class.

For example, Fig. 2 depicts a class called *Windows* which models the class of all video windows. The class *Windows* has four attributes. *Name* which is a string giving the name of each object instantiating *Windows*. *InfNames* is a set of strings (here modelling names of interfaces of an instance of *Windows*). *ReqQoS* and *ProQoS* denote the required and provided QoS, respectively, which both are QL clauses, as discussed in Section 3.3.

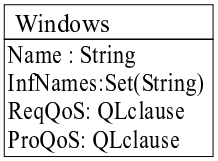

Figure 2: Example of a class

Attributes in our UML diagrams are always typed. Such type attributes can be predefined types, e.g., *Boolean*, *Integer*, *Real*, *Set*, *Sequence* ..., with their obvious intended meaning. We also need to use additional types which provide further structure, for example, we use a type QLclause as the type of *ReqQoS* and *ProQoS*. The main purpose of including types in UML models is to be able to use OCL expressions to describe constraints on classes and their behaviour, as will be described in Section 5.2.

Classes can be organised into a graph (or a collection of graphs), to build a *class diagram.* This describes their static relationship, the classes are linked together by *association* (a structural relationship that specifies the connection between one or more members of the classes) or *generalisation* (a relationship between a general class and a derived class which inherits its properties). Associations are depicted by simple lines. In order to denote how many instances of a class are related to an instance of the other class, at the end of the association the multiplicity of the association is included. Generalisations are depicted by directed lines with closed arrow heads.

For example, Fig. 3 denotes a class diagram comprising the class *Windows* and a second class of interfaces (*Infs*) which models the interfaces of an instance object from *Windows*. Instances of *Infs*, which are interfaces, have three attributes. *Name* is a string denoting the name of the interface. *InfType* is an enumeration. The type *enumeration*, denoted by *enum*, is an OCL type (see Section 5.2 below) which is used for attributes within UML models. The syntax of *enum* is defined as $\mathsf{enum} = \{\mathsf{value1}, \mathsf{value2}, \mathsf{value3}\}$ within the UML model. As a result, *InfType* determines if the interface is an operational interface *OpInf*, a stream interface *StrInf* or a signal

interface *SigInf*. The third attribute of *Infs* is the causality of the interface which is either a consumer *cons* or a producer *prod*. Two classes are connected together via an association which simply assigns an instance of *Windows* to one or more interfaces.

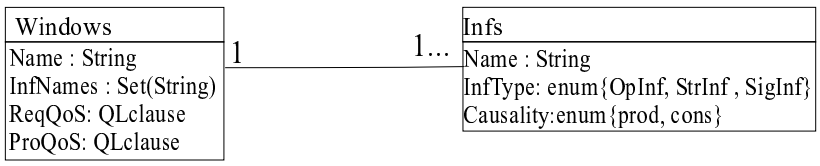

Figure 3: Example of a class diagram

A class diagram is like a template that demonstrates the static relationship between the system parts at a high level of abstraction. For example, suppose that a class diagram has been defined for a system which describes its components and relationships between them in general terms. Then, for a given example of a system, one can instantiate objects from the class diagram to create a model of the particular components and parts for the example. The relationship between classes from which objects are instantiated, is depicted in the class diagram. Subsequently, the relation between the components of the particular example automatically follows the pattern of the relationship between corresponding classes in the class diagram (see [36]). This results in a graph of instantiated objects and their connection patterns. The UML terminology for such an instantiated graph of objects is the *object diagram.*

For example, the class diagram of Fig. 3 has the pattern that a class *Windows* is connected to one or more instances of the class *Infs* denoting the interfaces. Now, consider a specific application containing a video window, an instance of the class *Windows* with the name *VidWin* which provides two interfaces to the distributed environment as follows. The first interface, called *VidWinIn*, is for a video input to the video window which receives the video frames created via some object (for example a camera) and transfered through a stream channel. The second interface, called *VidWinStat* is a signal interface representing the state of arrival of the video frames by emitting signals that the frame has arrived. Both above interfaces are instances of the class of interfaces *Infs* in the class diagram of Fig. 3.

The object diagram of Fig. 4 is instantiated from the class diagram of Fig. 3. To see this, note that we need to instantiate one object from the class *Windows* for *VidWin*. We also need to instantiate two objects *VidWinIn* and *VidWinStat* from the class *Infs*. *VidWinIn* is a stream interface supporting a video stream called *VidIn*. *VidWinStat* is a signal interface which supports two signals with signatures *VidPres* and *VidIn*, see the QL clause for provided QoS (*ProQoS*). *VidPres* is emitted to the environment when a video frame is presented from the *VidWin*. The interface *VidIn* could be used for further development of the model. For example, it could bind into a QoS manager to safeguard the QoS. On arrival of a video frame in *VidWin*, the interface *VidWinStat* receives a signal *VidIn*. Naturally the pattern of relations between the *VideoWindow* class and the *Infs* class follows and results in the creation of the object diagram of Fig. 4.

The object *VidWin*, instantiating *VideoWindow*, has a set of interface names and provided required QoS denoted by *ProQoS* and *ReqQoS*, respectively. The provided QoS is specified to provide, firstly, a throughput of at least 25 frames per ms into the interface *VidWinStat* and, secondly, there will be a maximum delay of 10 ms between the arrival of a frame and its subsequent presentation. For example, we shall explain

$$\forall\tau(VidWinStat.VidPres.SE, n) \leq \tau(VidWinStat.VidIn.SR, n) + 10$$

which referes to maximum delay of 10 ms. There are signals *VidPres* and *VidIn* assigned

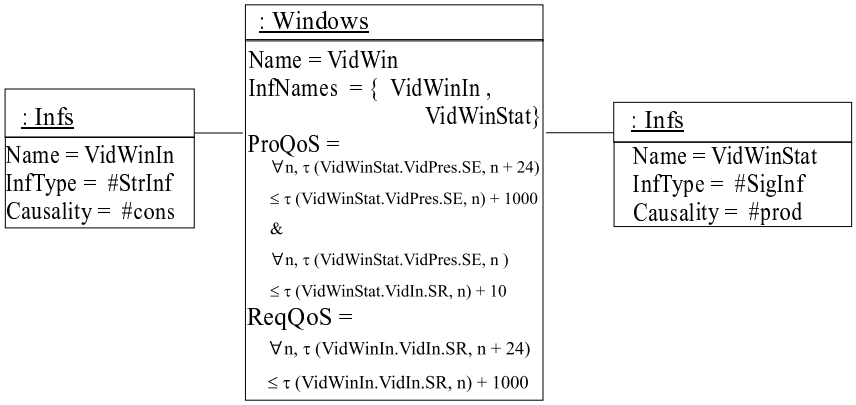

Figure 4: Example of an object diagram

to the interface *VidWinStat*. *VidPres* is emitted when a video frame is presented. Hence, $\tau(VidWinStat.VidPres.SE, n)$ denotes the time of presentation of the nth frame. The signal *VidIn* marks arrival of frames in the window to be displayed. Hence, $\tau(VidWinStat.VidIn.SR, n)$ denotes the time of arrival of the nth video. Consequently, the above QL clause reads as, the time of presentation of the nth frame is at most 10 ms after its arrival. In a similar way, we can see the required QoS for the object *VidWin*. *VidWin* requires a throughput of at least 25 frames per ms from its environment. In the UML, to distinguish between class diagrams and object diagrams, every object is underlined. For example in Fig. 4 (: Windows) denotes an object in instantiated from the class *Windows*.

In addition to the structural aspects of a specification (e.g., class and object diagrams), UML also provides notations to describe the behavioural aspect of the system under consideration. Such dynamic aspect diagrams come in a number of forms, and include state machines, use cases and collaboration diagrams [36].

Although the main thrust of this paper is to describe an approach to defining the static aspects of a system, as an illustration of the dynamic aspects Section 5.5 below models the dynamics of one of the components in our multi-media example.

4.1 Using OCL constraints

Although UML diagrams give a comprehensive account for many static and dynamic aspects of a systems design, there is often additional information that needs to be described in a class or object diagram for which UML on its own is not sufficient. However, many of these aspects can be described by using the Object Constraint Language (OCL) [37, 36], which is a textual notation for expressing constraints in UML diagrams. We will use OCL to describe a number of aspects including the types of attributes and the required relationships between QoS clauses.

The basic building blocks for OCL expression are objects and object properties which are instantiated from classes. In OCL, each object, attribute and operation has a type. Types in OCL are either *predefined types* or *user-defined* types. *Predefined* types are either a *basic type*, like *Integer*, *Real*, *String*, and *Boolean* or a *collection type* like *Collection*, *Set*, *Bag* and *Sequences*. Details regarding predefined types can be found in [36] and [37]. *User-defined* types are defined by UML models. For example, a class defines a new type. All instances of such a class are of its type. In fact, each class, interface, or type in a UML model is automatically a type in OCL.

One of the features of our work is the introduction of *QLclause* as a type for specifying required and provided QoS of computational objects or binding objects. This enables us to express the QoS aspects related to interaction of such objects with their environments.

In terms of constraints there are three types in OCL: *preconditions, postconditions* and *invariants*. Preconditions and postconditions deal with the behaviour of an operation in terms of its before and after states. In addition to these behavioural aspects there are sometimes certain constraints on the components of the system which can not be expressed in the class diagram. For example, we wish to express the relationship between the required and provided QoS on computational objects as part of our model of the computational viewpoint. OCL can be used to formalise such constraints in terms of *invariants* on the class diagram.

As the name suggests, invariants must be true at all times, and each invariant is declared by the keyword inv and has a context which refers to the class that the invariant must be applied to. For example, the expression

$$\mathsf{context}\ \ SomeClass\ \ \mathsf{inv}: SomeOCLExpression$$

simply says on a class called *SomeClass* the expression *SomeOCLExpression*, which is written in OCL, must be valid for all instances. To refer to instances of a class, OCL uses the keyword self. For example, if the class *SomeOCLExpression* in the above example has an attribute *SomeAttribute*, then self.*SomeAttribute* refers to the attribute *SomeAttribute* of all instances of the class *SomeClass*. For further information regarding the syntax and semantics of OCL, we refer the reader to [37] and [36].

Naturally, restrictions created by an OCL invariant are carried through to any instantiation of the class diagram into an object diagram.

5 A UML model of the computational viewpoint

The architecture of the UML [36] is based on a four layered structure of *user object*, *model*, *metamodel* and *metametamodel*. The metametamodel defines the language for specifying metamodel structure. A metamodel, which is an instance of the metametamodel, defines the language for specifying the model. Models, which are instances of metamodel, define the language to describe an information domain from which user objects, describing a specific application domain, are specified. In this paper we shall only be dealing with the three bottom layers user object, models and metamodels of the UML architecture.

Being inspired by the three layered structure *user object*, *model* and *metamodel* of UML, Fig. 5 depicts the analogy between UML architecture, computational viewpoint of RM-ODP and our UML model of the computational viewpoint. The left hand side column of Fig. 5 denotes the three layers of architecture as just discussed. The arrow between the layers denotes the direction of instantiation of one layer to the next. The middle column depicts the ODP approach. The upper layer "Computational viewpoint of RM-ODP" describes the computational aspects of the architecture for building open distributed systems. Such descriptions result in templates specifying the domain of application which basically models the computational aspects of group of applications with some common features. The "computational specification of a specific domain of application" can be specified with the help of the corresponding template. The arrow denotes the direction of moving from one stage to the next. The sign $\approx$ reflects upon similarity between the corresponding layer in the UML and RM-ODP analogy.

The right hand side column of Fig. 5 depicts our UML model that captures the static aspects of the ODP computational viewpoint. From the top row, we start by presenting a metamodel

UML		**RM−ODP**		**UML model of Computational viewpoint**
Metamodel	$\approx$	Computational viewpoint of RM−ODP	$\leftrightarrow$	CMD
$\downarrow$		$\downarrow$		$\downarrow$
Model	$\approx$	Computational specification of a domain application	$\leftrightarrow$	CCD
$\downarrow$		$\downarrow$		$\downarrow$
User object	$\approx$	Computational specification of a specific domain of application	$\leftrightarrow$	COD

Figure 5: Different layers of modelling

diagram of the ODP computational viewpoint, which for the purpose of brevity we shall refer to as the *Computational Metamodel Diagram* (CMD). The computational metamodel diagram includes meta-level classes in correspondence to different computational entities such as computational objects (*compobj*), bindings (*bindobj*), interfaces and reactive objects. The CMD also demonstrates the way that instances of such classes interact with one another.

The most important advantage of the CMD is that it provides a consistent method of modelling the QoS aspects in a computational viewpoint of a distributed system. In other words to model the static properties of the computational viewpoint of a given distributed system, one can instantiate the CMD to create a class diagram for a specific domain of distributed system that we shall refer to as the *Computational Class Diagram* (CCD). The computational class diagram is an instantiation of the CMD of the computational viewpoint of the corresponding distributed system. The CCD serves as a template for modelling static aspects of the computational viewpoint of applications which belong to the specific application domain that corresponds to the CCD. As a result by instantiating from the CCD, for each specific application, which belongs to the corresponding domain of application, an object diagram called the *Computational Object Diagram* (COD) is created which models the static aspects of the computational viewpoint of such application.

The CMD only describes the classes and their mutual relationship, as a result it is not capable of imposing conditions such as requirements for primitive bindings or the way that the requested and provided QoS of the components interact and the QoS of other parts of the system. Hence our model will include a set of OCL expressions to impose suitable restrictions on the CMD. Such restrictions naturally carry through to the instantiated CCD and COD of each distributed system domain and application.

The reminder of this section consists of four parts. First, we shall introduce the computational metamodel diagram. The second part includes OCL expressions imposed on the computational metamodel diagram. The third part introduces the computational class diagram and presents a heuristic for the construction a CCD for a specific domain of application from the CMD. The third part also includes an example of instantiating the CMD to create the CCD for an specific domain of application involving a video stream channel. The final part explains the creation of the COD model for the specific video stream example.

5.1 Computational Metamodel Diagram

The computational metamodel diagram, shown in Fig. 6, introduces meta-level classes (metaclasses) to represent computational objects, binding objects and reactive objects together with their associated interfaces.

The metaclass *SystemComponent* embodies classes of computational objects and binding objects. Similarly, the metaclass *ReactiveObject* represents the class of all reactive objects. The metaclass *Infs* represents interfaces to computational or binding objects, and the metaclass *RSigInfs* represents interfaces to the class of all reactive objects which is always a signal interface.

Between the metaclasses in the CMD there are a number of *associations*, and each association models the relationship between the classes that it connects. For example, the association between *SystemComponent* and *Infs* models interfaces assigned to computational or binding objects. Similarly, the association between *ReactiveObject* and *RSigInfs* models the signal interfaces of the reactive objects.

Each metaclass in the diagram has a name (e.g. *SystemComponent*) and a number of attributes. Consider, for example, the *SystemComponent* class. This class has five attributes. The attribute *Name* to identify the *SystemComponent*. The attribute *Role* is used to differentiate between computational objects (*compobj*) and binding objects (*bindobj*). The attribute *Role* is of type *enumeration* (*enum*) which lists the set of possible values. In order to use one of the values in an OCL expression, we have to prefix it with the # symbol.

Within the metaclass *SystemComponent*, the names of the interfaces of each computational or binding object are listed by the attribute *InfNames*. Finally, the attributes *ReqQoS* and *ProQoS*, which are clauses written in QL, correspond to required and provided QoS of a *SystemComponent* object as defined in Section 3.3.

UML objects are categorised according to the role that they play in the application. Interfaces in a distributed system are key ingredients as all interactions between objects are carried in the interfaces. As a result, in order to model interfaces, we represent an interface to a computational or binding object as a separate class *Infs* (along the lines of [15]). Each interface contains an attribute *Name* which gives the name of the interface and an attribute *InfType* which clarifies if it is an operational interface *OpInf*, a signal interface *SigInf* or a stream interface *StrInf*.

Each interface is in fact one of three types, and in the CMD this is represented by the use of three subclasses *OpInfs*, *StrInfs* and *SigInfs* corresponding to operational interfaces, stream interfaces and signal interfaces in the computational model, respectively. Thus, in UML terminology, the class *Infs* is a generalisation of classes *OpInfs*, *StrInfs* and *SigInfs*.

We start by explaining the metaclass *OpInfs* of operational interfaces. The RM-ODP [16] specifies that

> An operation interface signature comprises a set of announcement and interrogations signatures as appropriate, one for each operation type in the interface together with an indication of the causality (client or server, but not both) for the interface as a whole, with respect to the object which instantiates the template. *Clause 7.1.12.*

The metaclass *OpInfs* models the above by including a list of names of operational interface signatures (*OpInfSig*), and the causality of the interface (*Causality*). Assigned to each operational interface (*OpInfs*) there are metaclasses *Announcements* and *Interrogations* each one containing the attribute *NamesOfInv* which lists names of invocations as a set of strings. Such names by the clause 7.2.1 are "identifiers" of the corresponding signatures. Associations called *Ann* and *Inter*

depict the correspondence between *operational interfaces* and their corresponding *Announcements* and *Interrogations*.

Now, we shall deal with the metaclass *Announcements*. The RM-ODP [16] explains

> Each announcement signature is an action template containing both the names of the invocations and the number and names and types of its parameters. *Clause 7.1.12.*

This is modelled as a metaclass *Announcements* which contains the name of the invocations (*NamesOfInv*) which are identifiers of announcements by the clause 7.1.12. The metaclass *Invocations* has an attribute *Name* as an identifier and captures attributes of each involving invocation by including the number of parameters (*NumberOfPar*) and names of parameters (*NamesOfPar*). By clause 7.2.1 of [16], such names are identifiers for parameters in the context of each instantiation of such metaclass. Types of such parameters are instantiated from the metaclass *Parameters* which include the attribute *Type* of type *InfType*, which refers to the type of the parameters in the system. For example, in an application involving stream interactions *InfType* can be instantiated to include *audio*, *video* and *animation* referring to possible types of a parameter representing a stream interaction.

To explain the part of CMD dealing with *Interrogations* we cite part of the RM-ODP [16]:

> Each interrogation signature comprises an action template with the following elements:
> – the name of the invocations
> – the number, names and types of its parameters,
> – a finite, non-empty set of action templates, one for each possible termination type
> of an invocation, each containing both the name of the terminations and the number, names and types of its parameters. *Clause 7.1.12.*

As a result to model interrogations, we start by the metaclass *Interrogations* which contains names of all invocations (*NamesOfInv*). Each such invocation is instantiated from the metaclass *Invocations* which is identical to the metaclass *Invocations* of the metaclass *Announcements*. In this case, for each invocation there are a finite number of terminations, each one instantiated from the metaclass *Terminations*, which includes names of terminations (*NamesOfTerms*) as identifiers, see clause 7.2.1 of [16]. Each termination is instantiated from the metaclass *ATermination* which contains the name *Name* of the termination, the number of parameters (*NumberOfPar*) and the names of parameters (*NamesOfPar*).

In a similar way, the metaclass *StrInfs* corresponding to stream interfaces contains attributes *StrInfSig* and *Causality*. *StrInfSig* lists names of stream interface signatures which are "identifiers" of corresponding signatures. For each stream interface listed, there is an instance of the metaclass *Streams* with attributes *Name*, *NumberOfPar* and *NamesOfPar* referring to the numbers and names of the parameters. The metaclass *Parameters* models parameters and in fact is identical to the metaclass *Parameters* discussed earlier above.

The submetaclass *SigInfs* models signal interfaces. The RM-ODP [16] explains signal interfaces as follows.

> A signal interface signature comprises a finite set of action templates, one for each signal type in the interface. Each action template comprises the name for the signal, the number, names and types of its parameters and an indication of causality (initiating or responding, but not both) with respect to the object which initiates the template. *Clause 7.1.12.*

This is modelled as a metaclass *SigInfs* of signal interfaces which contains signal interface signatures (*SigInfSig*) expressed as a set of names of signals which by clause 7.2.1 are identifiers of such signatures. Each signal interface is instantiated from the metaclass *Signals* which includes the name (*Name*)of the signal, the number of parameters (*NumberOfPar*), names of parameters (*NamesOfPar*) and causality (*Causality*) of the signal.

The main difference between signal interfaces and operational and stream interfaces is that, in the case of signal interfaces, the causality applies to each one of the (possibly many) signals of an interface, while the RM-ODP defines the causality as a whole for operational and stream interfaces. As a result, *Causality* appears as an attribute in metaclass *OpInfs* and *StrInfs* while in case of signals the attribute *Causality* appears in the metaclass *Signals* which models individual signals of a signal interface.

The metaclass *ReactiveObject* in the CMD embodies all reactive objects. A reactive object in the computational viewpoint can only have signal interfaces. Hence, the only attribute of the metaclass *ReactiveObject* are names (*Name*) of the reactive objects and a list of signal interfaces *SigInfNames*. Each such interfaces is instantiated from the metaclass *RSigInfs* containing the names of signals (*SigInfSig*) which are identifiers for the signals. Each signal is instantiated from the metaclass *Signals* which was explained earlier.

The RM-ODP [16] introduces two types of binding, *compound* binding and *primitive* binding. The compound binding action of the RM-ODP is performed by a binding object. As discussed earlier, in the CMD the compound binding object is an instantiation of the class *SystemComponent* with the role of *bindobj*. A primitive binding action allows binding of two interfaces of the same or different computational objects. The primitive binding between interfaces (*Infs*) is denoted by the self association called *PB* on the metaclass *Infs*. Similarly, to model the primitive binding between signal interfaces of the reactive object, *RSigInfs*, and the corresponding signal interface of the system that the reactive object is to be bound to, an association between metaclasses *RSigInfs* and *SigInfs* is created and is referred to as *Rsigbind*.

5.2 OCL constraints on the class diagram

As mentioned above, a number of constraints need to be placed over the metaclass diagram. We discuss these in turn now.

In the CMD model of Fig. 6, the primitive binding actions instantiate the association from class *Infs* into itself. The RM-ODP requires a primitive binding to be between interfaces of the same type and with complementary causality. To represent this we impose the following OCL invariant on the CMD.

```
context  Infs
 inv :     self.PB.InfType = self.InfType
 inv :     self.PB.Causality <> self.Causality
```

As discussed earlier the computational viewpoint adopts a contractual approach to the issue of QoS. In particular, each object can provide the specified QoS only if it receives its required QoS from its environment. We shall impose this as an invariant on the *SystemComponent* metaclass in the CCD as follows.

```
context    SystemComponent
```

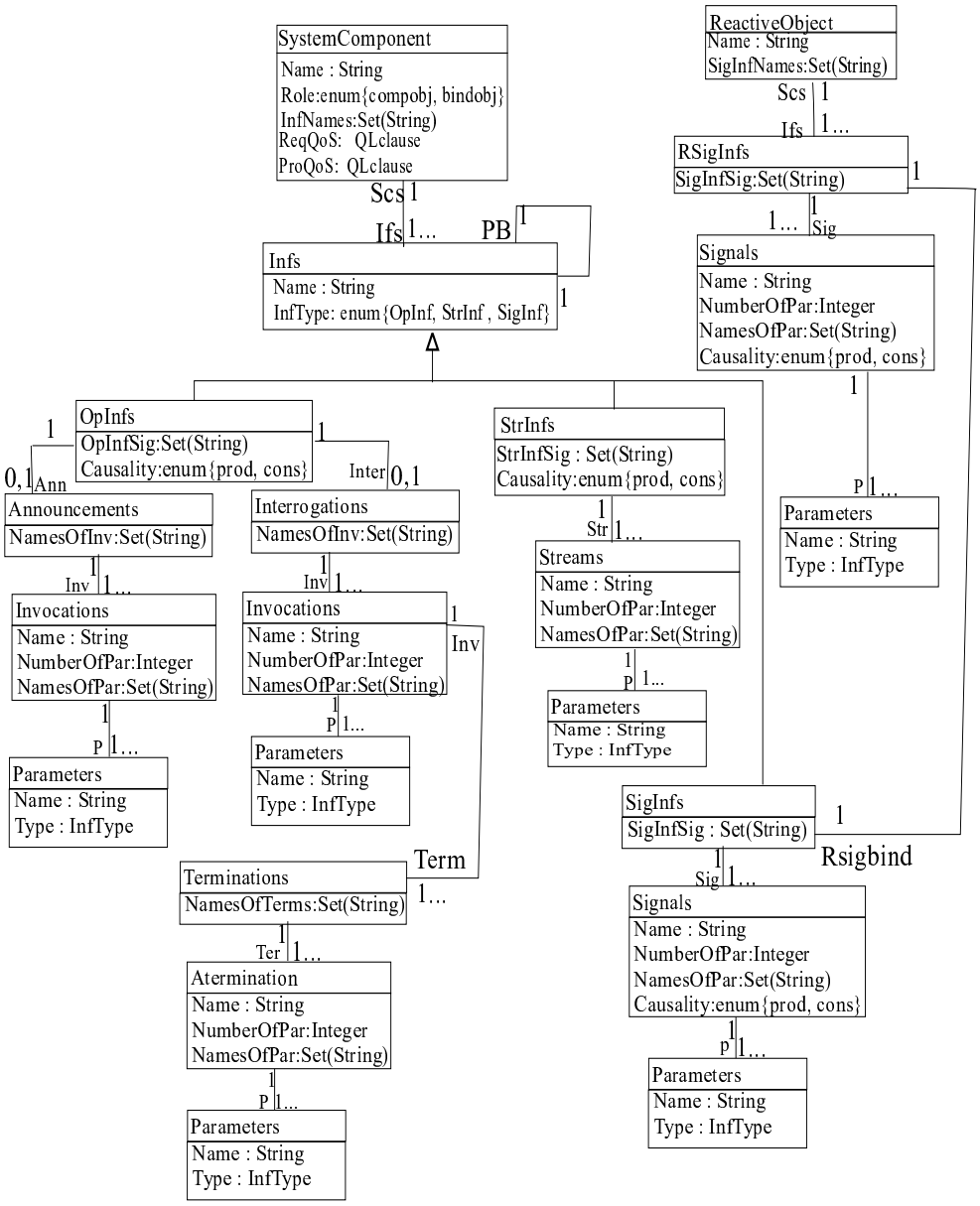

Figure 6: The Computational Metamodel Diagram

```
inv :   if self.ReqQoS = true then self.ProQoS = true
        else self.ProQoS = false
        endif
```

The RM-ODP states that operations are either *announcements* or *interrogations*, but not both. The following implements this restriction with two invariants, the first one states that the set of all operational interface signatures (self.OpInfSig) is the union of the set of corresponding announcements self.Ann.NamesOfInv and interrogations self.Inter.NamesOfInv. The second invariant states that the above sets of announcements and interrogations are disjoint.

```
context   OpInfs
 inv :   self.OpInfSig = (self.Ann.NamesOfInv -> union(self.Inter.NamesOfInv))
 inv :   self.Ann.NamesOfInv -> intersection(self.Inter.NamesOfInv) -> isEmpty.
```

The attribute *NamesOfInv* of the class *Announcements* lists the names of all invocations. We need to instantiate exactly those invocations which belong to the set of names of invocations *NamesOfInv* of the class *Announcements*. The following OCL invariant states that the set of all *NamesOfInv* listed under *Announcements* is identical to the set of names of all instantiated invocations (self.Inv.Name).

```
context   Announcements
 inv :   self.NamesOfInv = self.Inv.Name.
```

We need to include similar OCL constraints in context of classes *Interrogations*, *Terminations*, *StrInfs*, *SigInfs* and *RSigInfs*.

The class *Invocations* which models invocations includes the names of parameters (*NamesOfParameters*) as an attribute. Each parameter is instantiated from a class *Parameters*. We need to instantiate exactly as the same *Parameters* object as declared in its corresponding *Invocations* object. This is explained by the following invariant.

```
context   Invocations
 inv :   self.NamesOfPar = self.P.Name.
```

In a similar way we must write invariants on all other classes which are linked to the class *Parameters*, for example *Invocations*, *Atermination* and *Streams*.

5.3 Computational Class Diagram

This section deals with creation of the *Computational Class Diagram* (CCD), a UML class diagram, which captures the static aspects of the computational viewpoint of a *Specific Domain of Application* (SDoA). As depicted in Fig. 5, the CCD of a specific domain of application firstly must be instantiated from the CMD and, secondly, must present a model of the computational viewpoint of its corresponding domain of application. Finally, it must provide us with a template from which all possible applications satisfying the description of such a SDoA can be instantiated into an object diagram.

For example, consider video stream channels a necessary component of the lip synchronisation mechanism as a specific domain of application. This SDoA normally consists of one or more cameras for capturing images to be transfered over stream bindings into one or more display windows and there are various control mechanisms to operate the components and safeguard the QoS. Now, an application consisting of one fixed camera with no control for movement or zoom, one window and a binding which provides a degree of QoS, is an instance of such SDoA. Examples of instances of this SDoA also include the following:

1. An application consisting of a camera with an operating mechanism to rotate, zoom and tilt which also imposes a more demanding set of QoS restrictions.
2. An application consisting a number of cameras capturing pictures from different parts of an area whose outputs are presented via different windows, say in a Closed Circuit TV system.

The purpose of the CCD corresponding the above SDoA is to act as a template from which all the above instantiated examples (which can be modelled as object diagrams) can be created.

At this stage we explain a heuristic for the creation of the CCD of a SDoA from the CMD of Fig. 6. In what follows the word *metaclass* refers to the CMD of Fig. 6.

1. Start by instantiating classes from the metaclass *SystemComponent*. In doing so, new attributes and operations related to each class are included to specialise each class as a template model of a specific part of the system. The main criterion in creating a new class is whether we need to include new attributes in the class which are not pertinent to the existing classes.
2. Each class created in the previous step must be connected to an instance of an interface metaclass *Infs* which also can include new attributes. According to the SDoA that we are trying to model, we might need to instantiate one or more interface classes. This is obviously a matter of modelling practice and can be judged by whether there are enough distinguishing attributes of operations that result in creation of new interface classes.
3. Subclasses of each interface class must be instantiated from the corresponding subclass of the metaclass *Infs*.
4. Different control mechanisms are modelled as reactive object classes instantiated from the metaclass *ReactiveObject* including the required attributes and operations related to the SDoA.
5. Following the pattern in the CMD, for each instantiated *ReactiveObject* class, one or more signal interface classes (which are instantiated from the class *RSigInfs*) are created. In each case associations between metaclasses are instantiated to connect suitable classes.
6. In the same spirit as the third step of the heuristic, subclasses of all created instances in the previous step must be created and suitably connected to instantiate the corresponding association.

Fig. 7 denotes the CCD of the SDoA of the video stream channels as discussed above. Following the above heuristic, in this SDoA there are four distinguishable classes *Cameras*, *Windows* and *StreamBinding* instantiated from the metaclass *SystemComponent* and *QoSBindings*, which is instantiated from the metaclass *ReactiveObject*. At this stage specific necessary attributes and operations are added. For example, the attributes *panDegree*, *tiltDegree* and *zoomFactor*, which are integer valued attributes referring to degrees of pan, tilt and zoom of each camera are included.

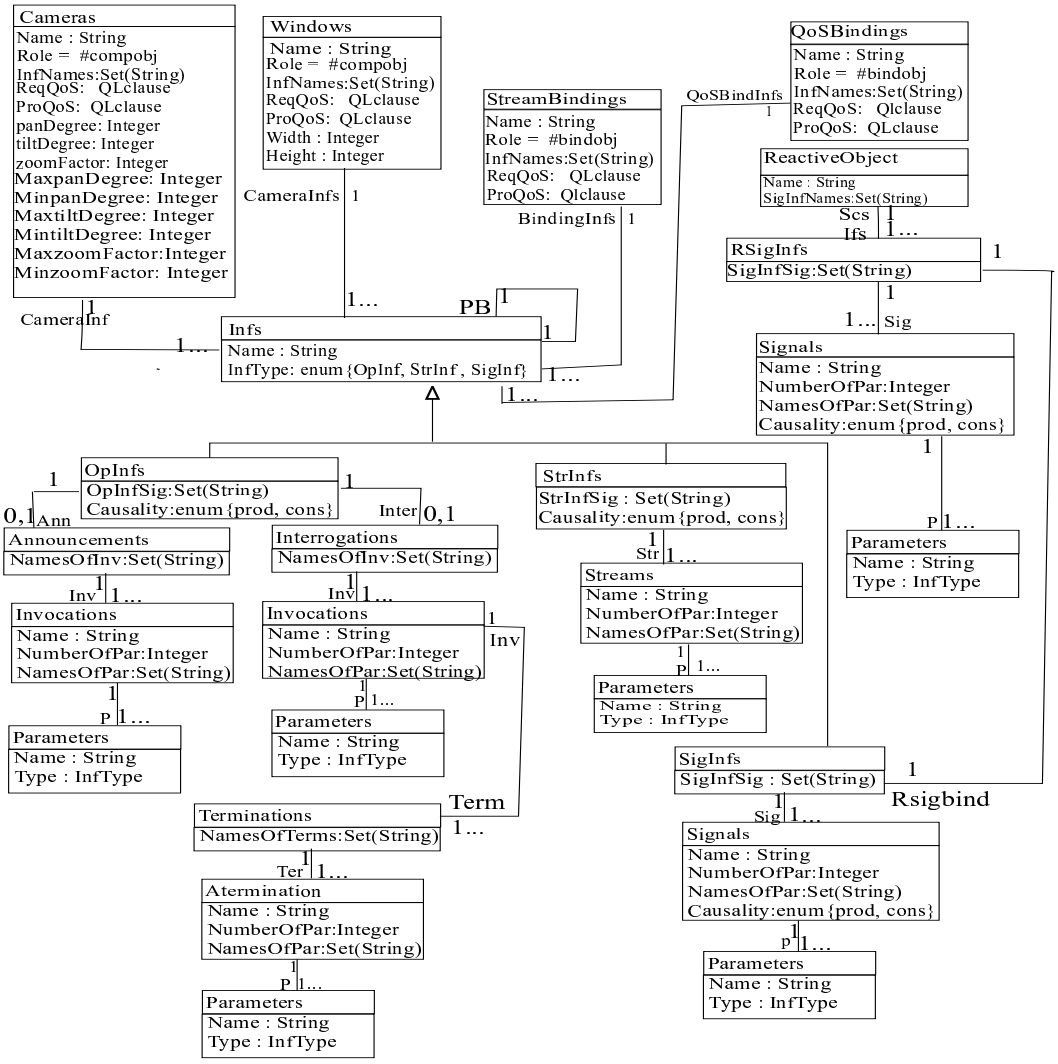

Figure 7: The CCD of the video stream channels

Also, attributes related to the maximum and the minimum of pan degrees, tilt degrees and zoom factors are added to the class *Camera*. Attributes *Width* and *Height* of window displays are added to the class *Windows*.

The next stage is the instantiation of interface classes from the metaclass *Infs*. In this SDoA, there is no ground for the instantiation of more than one class from the metaclass *Infs* and we don't consider any new attributes or operations to be added. As a result, there is only one class *Infs*. Similarly, submetaclasses of the metaclass *Infs* are instantiated to the corresponding subclasses of the newly created class *Infs* identically. In order to simplify the model we have included only one type of reactive object.

5.4 Computational Object Diagram of a video stream

As the final step in our modelling method, in this section we shall deal with the creation of Computational Object Diagram (COD) from the Computational Class Diagram of a Specific Domain of Application. A COD, which is a UML object diagram, depicts the static aspects of the computational viewpoint of an application which satisfies the description of a SDoA. As depicted in Fig. 5 each COD is instantiated from the corresponding class diagram.

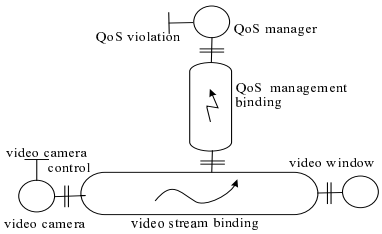

Figure 8: Video stream channel

For example, consider the video stream channel of Fig. 8 comprising of a *video camera* which produces audio frames to be carried over a *video stream binding* into a *video window*. The video camera is controlled and operated by a *video camera control* mechanism and the QoS of the stream binding is safeguarded by a *QoS manager* which interacts with the video stream binding through a signal binding of *QoS management bindings*. In case of violation of QoS, the interface *QoS violation* emits suitable signals to alert the environment.

The current example obviously complies with the description of the SDoA of the previous section, which is modelled via the CCD of Fig. 7. The CCD of Fig. 7 acts as a template for creation of the COD of our example. The heuristic for the generation of the COD is as follows, in which the word class refers to the CCD of Fig. 7.

1. Start by instantiating computational objects from the classes with the role *compobj*.
2. Instantiate all reactive objects, from the classes *ReactiveObject*.
3. For each object created in steps 1 and 2 create object interfaces by instantiating objects from the classes *Infs*.
4. For each created interface in step 3, instantiate all its associated objects. For example, consider an object which instantiates a stream interface class (*StrInfs*). Such object includes stream signature names under the attribute *StrInfSig*. For each such signature we need to create an object instantiated from the class *Streams*, which must be connected to the

StrInfs object. Subsequently, for each instantiated object *Streams*, we must also instantiate parameter objects from *Parameters* classes to model parameters.

5. In order to model primitive bindings, which allows binding of two interfaces, connect corresponding interface objects via associations and mark them with *PB*.

6. When there are primitive bindings between interfaces (associations which are marked with *PB*) we need to clarify which parameters of one interface map into which parameters of the other interface. Connect corresponding parameter objects via an association and mark them with *map* to denote that they map together.

Applying the above heuristic to the example of Fig. 8 produces the COD of Fig. 9. As a result, there are five main objects with *Name* attributes of *Camera1*, *VidWin*, *VidBind*, *QoSManBind* and *QoSManger* instantiated from the classes *Cameras*, *Windows*, *StreamBinding*, *QoSBinding* and *ReactiveObject*, respectively (see Fig. 7). Each of the above objects has the corresponding attributes of the class that it is instantiated from and we shall only explain the QL clauses stating the required and provided QoS of each object. The object *Camera1* which models the video camera of Fig. 8 requires no degree of QoS from the environment, while it can provide at least 25 frames per second. The *Camera1* object also includes a list of attributes such as the *panDegree* which is declared as 35 degree. For the purpose of brevity we have elided some of the attributes.

The object *VidWin* models the video window and requires to receive at least 25 frames per second, but it provides no QoS to the environment. The video stream binding is modelled via the object *VidBind* which provides a throughput of 25 frames per second and a delay of between 40 and 60 ms. Also, *VidBind* guarantees that the monitoring signal will be emitted through the *QoSCtrl* interface at the time of the occurrence of the monitored signal. The *VidBind* object requires that it receives at least 25 frames per second from the producer. The QoS of management binding of Fig. 8 is modelled as the object *QoSManBind* instantiated from *QoSBindings*. *QoSManBind* needs no QoS from the environment and it can guarantee a maximum delay of 5 ms for the transition of signals across the binding.

The object *Camera1* has two interfaces listed under the attribute *InfNames* as *VidCamOut* and *CamCtrl* which refer to the stream output interface into video stream binding and the interface for controlling and manipulating of the video camera, respectively. The interface object *CameraCtrl* embodies the operational interface signatures *start*, *stop*, *pan*, *tilt* and *zoom* referring to the corresponding actions of the camera object. These five operations are all announcements as a result they are embodied in the object :Announcements. The operation *start* being an announcement has an invocation which is modelled as a object called *start* which has no parameters. The operation *pan* is also modelled as an invocation object called *pan*. It has one parameter called *panDegree* which denotes the degree of pan that we would like to achieve by invoking the action *pan*. *panDegree* as a parameter is modelled via an object :Parameters embodying its name and type and is connected to the object *pan* via an association. The other three operations (*stop*, *tilt* and *zoom*) can be explained similarly.

The interface that video camera offers for binding to the video stream binding is modelled via the object *VidCamOut* which is a stream interface type *StrInf* and support a stream with the interface signature *VidOut* which is modelled as an object called *VidOut*. This object which is instantiated from the class *Streams* has one parameter called *video1*. The parameter *video1* is modelled as a :Parameters object which includes its name *video1* and its type *video*.

The interface *VidCamOut* of the object *Camera1* binds to the interface *VidBindIn* of the video stream binding. To model the primitive binding an association marked, as "*PB*", is used to connect *VidCamOut* and *VidBindIn*.

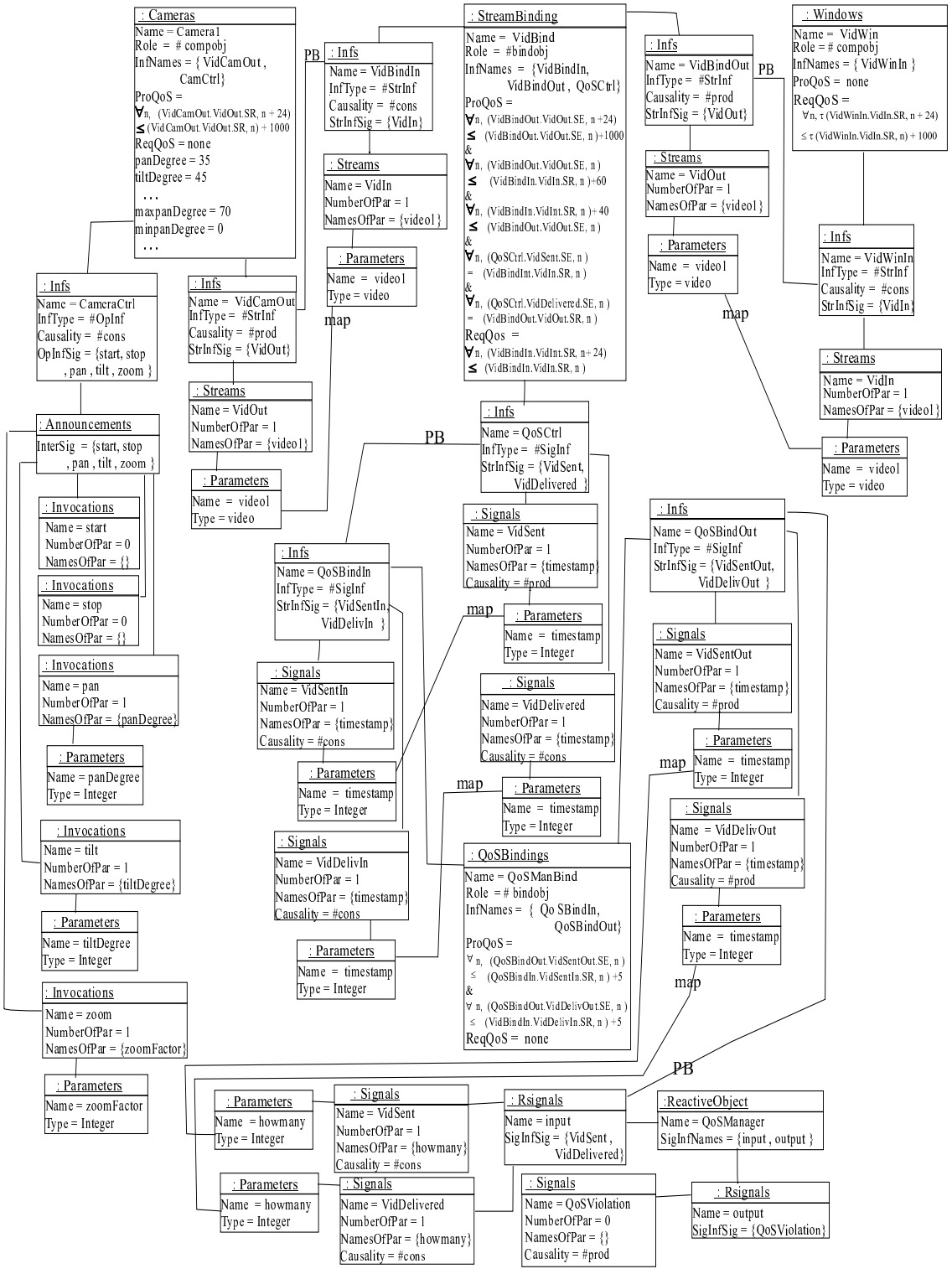

Figure 9: Computational Object Diagram for the video stream

There is also a primitive binding between the QoS management binding object and the video stream binding object which is modelled as an association between objects *QoSBindIn* and *QoSCtrl*. As there is more than one signal in each of these interfaces, corresponding parameter objects (:Parameters) are connected together via associations marked by the word "map".

The Computational Object Diagram of Fig. 9 specifies required and provided QoS statements of the components. There is a global requirement (expressed by an OCL constraint of section 5.2) that each of the components can provide its specified provided QoS as long as its required QoS is satisfied. We must emphasis that, we do not assume the QoS required by an object A from another object B is always identical with the QoS offered by B to A. We assume a more complex specification pattern that the QoS provided by A *implies* the QoS required by B. Therefore, if QoS required and provided by different objects don't match, we can see inconsistencies within the specification of the system.

We have now completed the production of our object model for this particular example, which models the static aspects of the system. The UML also provides a set of diagrams for the specification of the dynamic aspects of systems. One of the dynamic aspects of the above example is the implementation of the QoS aspects, which models the behaviour of the reactive object *QoSManager* in the COD. The next section shows how the behaviour of the reactive object could be described.

5.5 Modelling the behaviour

In order to specify the behavioural aspects of the reactive object *QoS Manager* of Fig. 8, we shall use the UML *State Machine Diagrams* [36]. The UML state machine diagram is an object-based variant of Harel's *Statecharts* [19], and we shall refer to that simply as *statecharts*.

Assume that the *QoS manager* of Fig. 8 modelled as the object *QoSManager* in Fig. 9 requires that the reactive object check for the following behaviour

- a throughput of at least 25 frames per sec and
- a latency of between 40 and 60 ms.

Moreover assume, in case of violation of any of the above requirements, the *QoSManager* alerts the system by emitting a *QoSViolation* signal.

The statechart of Fig. 10 depicts the above behaviour. It consists of two statecharts A and B. We shall explain statecharts A, B and the implication of the dotted line separating them in order.

The statechart A deals with the implementation of the throughput of at least 25 frames per sec. The two rectangular nodes marked by *Begin* and *Wait* are called *states* of the statechart A. The state *Begin*, which marks the start of the cycle of behaviour of A, is referred to as an *initial state*. Initial states are recognisable from other states by the distinctive sign of a black dot which is attached to them via an arrow.

There are several other arrows between states of A, which depict the way that states of A alter between *Begin* and *Wait*. Such arrows are called *transitions* and when state changes we say the corresponding transition has been *fired*. There are pieces of text assigned to transitions which includes both conditions and consequences of firing the transition. Such text is often of the form *Name:Event[Guard]/Action. Name* is a string denoting the name of the transition, and is merely a label intended for the individual statechart. *Event* is a specification of observable behaviour, for example, observing a signal. Events are triggered by the environment, and might

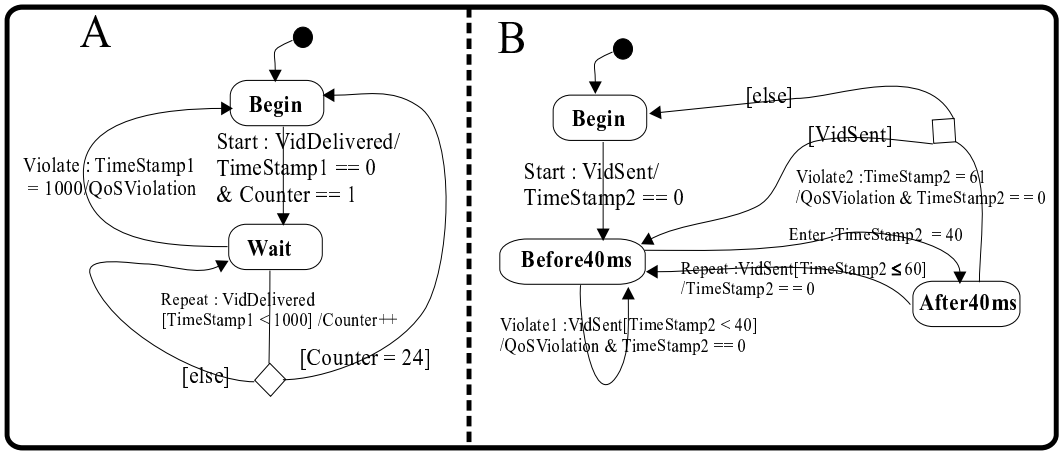

Figure 10: The statechart expressing the dynamic behaviour of QoS manager

include associated conditions. For example, the event *TimeStamp = 61* requires the value of the clock *TimeStamp* to be 61 for the behaviour to be enabled. *Guard* is a Boolean expression. Whenever *Event* is observed and, at the same time, *Guard* evaluates to *true*, the transition must fire immediately. Firing of a transition causes both a change of state and also an occurrence of all actions listed in *Action*. Actions consists of a list of events that will be triggered, and optionally assignment to the variables.

For example, in A, the transition from *Begin* to *Wait* is marked by *Start:VidDelivered/TimeStamp1* $==$ 0& *Counter* $==$ 1. As a result, the transition *Start* can fire as soon as the signal *VidDelivered*, marking the delivery of a single frame, is observed. In this case if *Start* fires, apart from a change of state from *Begin* to *Wait*, a clock *TimeStamp1* starts and is set to zero. Also, a counter *Counter*, which is an integer value variable, starts by assigning its value to 1. We assume that clocks are ticking at the rate of 1 ms and the type of a clock value is referred to as simply by TIME, hence *TimeStamp1* is of type TIME.

Firing of the transition *Start* changes the state of A to *Wait*. We will use this state to idle for 1 second to see if the rest of the 24 frames are delivered, i.e. the throughput is 25 frames all together. While in state *Wait*, if the time increments to 1000 ms, *TimeStamp1* $=$ 1000, then the transition *Violate* of A will immediately fire. This results in the emission of the action *QoSViolation*, a change of state to *Begin* and a start of a new cycle of behaviour of A. However, while A is in the state *Wait* and the time has not passed beyond 1000 ms (*TimeStamp1* $<$ 1000) and a frame is delivered i.e., *VidDelivered* occurs, then the transition *Repeat* of A can fire. When *Repeat* fires, the counter is incremented by one (*Counter* $++$) to keep a record of how many frames are delivered.

At this point there are two possible scenarios. The first scenario is that the number of delivered frames is 24, i.e. the guard [*Counter* $=$ 24] is *true*, which says that within the last 1 second after the delivery of the first frame 24 frames are delivered and the system is complying to the condition of a throughput of 25 frames per second. In this case, the state changes to *Begin* which marks the start of a new cycle of behaviour. The second scenario is that the number of the delivered frames is less than 24 and we need to return back to state *Wait* and wait for the delivery of the rest of the frames. The diamond shape node which depicts the above two scenarios of the behaviour of the transition *Repeat* is called a *conditional connector*.

The statechart B deals with the issue of latency. To have a latency between 40 and 60 ms, as

soon as the first frame is sent, observing *VidSent*, the state of B changes from the initial state *Begin* to the state *Before40ms*. In addition, a clock called *TimeStamp2* is reset to zero. The state remains in *Before40ms* as long as *TimeStamp2* is less than or equal 40 ms. Within this time interval if another frame is sent, observing *VidSent* again, then the latency condition is violated and the transition *Violate1* fires. This results in an emission of the signal *QoSViolation* and resets *TimeStamp2* to zero. While in state *Before40ms*, if no *VidSent* occurs, the time progresses to 40 ms ($TimeStamp2 = 40$) which results in firing of the transition *Enter* and arrival in the state *After40ms*. In the this state, while $TimeStamp2 \leq 60$, if *VidSent* is observed, then no latency condition is violated and the transition *Repeat* fires immediately. Firing *Repeat* resets the *TimeStamp2* to zero to deal with the latency associated with the new frame. However, assume that while in *After40ms*, the time progresses to 61 ms and no *VidSent* occur. Then within the interval time of 40 and 60 ms of the occurrence of the last *VidSent* no frame is sent, which is a violation of the latency condition. Consequently, *Violate2* fires immediately and this emits a *QoSViolation* and resets the *TimeStamp2* to zero. In this case, if *VidSent* is observed we return back to the state *Before40ms* to start a new cycle of behaviour related to the observed *VidSent* and if *VidSent* is not observed, we return back to *Begin* and wait for the occurrence of the next *VidSent*.

The statecharts A and B have disjoint sets of events, clocks and counters. As a result firing of transitions in one of them does not effect the behaviour of the other one. As a result, A and B are executed independently and not concurrently. This type of independent parallelism is denoted by placing the two statements side by side, separated by a dotted line, as Fig. 10 illustrates.

A full description of the behaviour of the video stream mechanism and of the full lip-synchronisation system is beyond the scope of the paper, as is a description of use cases, collaboration diagrams or other means of specifying the behaviour within the UML. Introduction to use cases, collaboration diagram and other UML diagrams are contained in [36, 8, 15].

6 Related Work

One of the objectives of the current work is to present a UML based method of specification of QoS constraints. Since the rise of UML, the World Wide Web Consortium [38] has proposed several means such as the Resource Description Framework (RDF) [29] and Composite Capabilities/Preferences Profiles CC/PP [10], which seem suitable for specifying QoS constraints on the models. For example, one could imagine using RDF for developing the metamodel of section 5. RDF can easily be serialised in eXtensible Markup Language (XML) [39]. Instances of such models could then directly be used by distributed systems for exchange, negotiation and configuration purpose. This can be a subject of further research, specially for the static aspects of the model. However, our approach, using UML, enables us to model more complex structural and behavioural interactions between objects.

Our approach to UML modelling of distributed multimedia systems is influenced by COMET [15]. Concurrent Object Modelling and architectural design mETthod (COMET) is a UML methodology which is tailored specifically for concurrent and distributed systems. COMET introduces the notion of *interface* classes and objects encapsulating external interaction of the components of the distributed system with the environment. We found the notion of interface objects of COMET suitable for describing interfaces in the computational viewpoint of RM-ODP.

Various approaches which integrate the UML and RM-ODP have been described in the literature. Oldewick and Berre [35] introduce a methodology based on RM-ODP and UML aimed at systems such as geographical information systems. Linington [24] uses UML to specify enterprise

specifications of the RM-ODP. Steen and Derrick [31] present a metamodel UML core for the enterprise viewpoint in the RM-ODP and also study the extent to which UML can be applied for the specification of the enterprise viewpoints. Aagadel and Milosevic [2] discuss different enterprise modelling concepts in terms of the UML and the way that enterprise viewpoints can be used as a part of the software development cycle via the UML.

Some researchers have applied UML to revisit distributed applications which were originally modelled based on the architecture of the ODP. Cornily and Belunde [9] define an ODP viewpoint architecture of the process ITU-Recommendation G.851-01 [14] within the context of telecommunications management networking. Kande et al. [20] apply the UML notations to specify service components of telecommunications management networking systems which are originally modelled in the framework of RM-ODP.

The CMD of Fig. 6 introduces a metamodel for describing families of classes. The UML also provides *stereotypes* as an extension mechanism to model a family of classes. A stereotypes [36] is denoted by a text between guillemets. For example, to declare that a class is an *interface* we can decorate it with << interface >>. Although introducing stereotypes to UML models seems a convenient modelling practice, it can cause certain ambiguities. For example, the exact semantic of the stereotype is not clear. Also, the issue that stereotypes are not modelling constructs against which analysis can be performed is a controversial one. Recently, Pinet and Lbath [26] have presented a semantic of stereotype for type specification. We have avoided using stereotypes, which remain areas for future research.

In this paper we use the QL, which has proved to be a useful and strong language for specification of QoS [12, 13, 30]. We have been able to to specify latency, jitter and throughput via QL. QL is not the only language used for the specification of QoS. Aagadel [1] presents ODL-Q which is based on TINA ODL [33] to specify the QoS characteristics of the underlying model. While this work is valuable, [1] uses the OCL as a formal language, separate from UML, for extending TINA ODL. As opposed to this, we specify QoS as attributes of objects in a UML model and use OCL statements to impose constraints on diagrams. Using two different languages of QL and OCL seems a bit inconvenient, and it would be desirable to make use only one of them. Unfortunately, neither QL nor OCL is strong enough to replace the other. QL, which is a temporal logic, does not have any mechanism for imposing constraints or navigating UML class diagrams. OCL, which is designed to enable navigating UML class diagrams and imposing constraints on them, does not have a time function to capture temporal aspects such as the time of occurrence of the events.

Extensions of QL have also been considered. For example, based on linear temporal logic, L. Blair [5] introduces QTL (QoS Temporal Logic). To specify stochastic aspects of QoS such as the *bounded responsiveness* property, QTL is extended to SQTL (Stochastic QoS Temporal Logic) by Lakas et al. [22]. These are more expressive than QL. However, our purpose was to illustrate the use of such a (real-time) logic within the UML diagrams, and it would be easy to extend the work presented here by incorporating other similar logic or approaches to QoS specification.

7 Conclusion

In this paper, we have used UML to build up a metamodel for the computational viewpoint of RM-ODP. We have incorporated QoS aspects of the model as attributes of metaclasses. Further constraints on the metaclasses are imposed by a set of OCL invariants on the metamodel. For example, one of our invariants imposes the contractual approach of the RM-ODP to QoS, which states that a component of the model can provide the specified QoS only if it receives its required QoS from the environment.

The computational viewpoint of RM-ODP acts as a guideline for modelling computational aspects of a distributed system. In the same fashion, the metaclass diagram can be instantiated to create a class diagram modelling static computational aspects related to a specific domain of application. In such class diagrams we can add further attributes and methods. Class diagrams act as templates from which objects diagrams modelling each individual application can be instantiated. We have demonstrated our approach with the help of an example of a video stream channel. In order to capture part of the dynamic of the video stream channel, we have modelled the control mechanism that governs transmission and safeguards the delivery of the QoS of the video frames as a statechart.

The RM-ODP has a broad scope, including support for all types of traditional data processing systems, real-time systems and multimedia systems. In future, we would like to implement our approach by modelling examples of such systems. Currently, our method only deals with the computational viewpoint, we are planning to extend our approach to incorporate other viewpoints of ODP in order to give a more refined model of the distributed system. Finally, there is further work to be done on behavioural aspects and their relationship to the approach presented here. Although, we briefly mentioned the specification of the associated behaviour in section 5.5, there is clearly scope for investigating how to link UML behavioural diagrams into the UML static structure that we have presented here. Regardless of QoS issues, studying the relationship between static and dynamic specification in UML models is an active area of research [21].

References

[1] J. O. Aagedal, *Towards an ODP-compliant Object Definition Language with QoS-support*, Presented at Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS98), September 1998, Oslo, Springer LNCS 1483, pp. 183-194, 1998.

[2] J. O. Aagedal and Z. Milosevic, *ODP Enterprise Language: UML Perspective*, 3rd International Enterprise Distributed Object Computing Conference (EDOC'99), Mannheim, Germany, pp. 27-30, 1999.

[3] T. Bolognesi and E. Brinksma, *Introduction to the ISO specification language LOTOS*, Computer Networks and ISDN Systems, 14 (1) pp. 25-59, 1988.

[4] G. Barry and G. Gonthier, *The ESTREL synchronous Programming Language: design, semantic, implementation*, INRIA report no. 842, Domaine de Voloceau-Rocquencourt, BP 105, 78163 Le Chesnay Cedex, France, 1988

[5] L. Blair, *The formal specification and verification of Distributed Multimedia System*, PhD thesis, Lancaster University, UK, 1994.

[6] G. Blair and J.- B. Stefani, *Open Distributed Processing and Multimedia*, Addison-Wesley, 1997

[7] G. Booch, *Object-Oriented design with applications*, 2nd ed. Reading, Mass., Addison-Wesley, 1994.

[8] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modelling Language user guide*, Addison-Wesley, 1998.

[9] J.-M. Cornily and M. Belaunde, *Specifying Distributed Object Applications Using the Reference Model for Open Distributed Processing and the Unified Modelling Language*, Proc. of The 3rd International Conference on Enterprise Distributed Object Computing (EDOC99), 1999.

[10] *Composite Capabilities/Preference Profiles* , W3C CC/PP working group, www.w3.org/Mobile/CCPP

[11] G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems - Concepts and Design*, Addison-Wesley, 1994.

[12] I. Demeure, L. Leboucher, N. Rivierre and F. Singhoff, *Modle et plate-forme pour le support d'applications multimdias rparties*, Confrence Francaise sur les systmes d'exploitation (CFSE 99), pp. 97-108, Rennes, 1999.

[13] I. Demeure, F. Singhoff and F. Horn, *Automatic Scheduling of a Dynamic Multimedia Applications with Polka : a Case Study*, Fourth IEEE Real-Time Technology and Applications Symposium (RTAS 98), Denver, Colorado, USA June 3-5, pp. 15-19, 1998.

[14] *ITU-T Recommendation G.851-01: Management and Transport Networks Applications of the RM-ODP framework*, 1996.

[15] H. Gomaa, *Designing Concurrent, Distributed and Real-Time Applications with UML*, Addison-Wesley Object Technology series, 2000.

[16] *ITU Recommendation X.901-904 ISO/IEC 10746 1-4*. Open Distributed Processing Reference Model - Parts 1-4, July 1995.

[17] I. Jacobson, *Object-Oriented Software Engineering*, Reading, Mass., Addison-Wesley, 1992.

[18] F. Jahanian and A. K.-L. Mok, *Safety Analysis of Timing Properties in Real-Time Systems*, IEEE Trans. Software. Eng., vol. 9 No. 12, pp. 890-904, 1986.

[19] D. Harel, *On Visual Formalisms*, CACM 31, no. 5 pp. 514-530, 1998.

[20] M.M. Kande, S. Mazaher, O. Prnjat, L. Sacks and M. Wittig, *Applying UML to Design an Inter-Domain Service Management Application*, UML 98, LNCS 1618, pp. 200-214, Springer, 1999.

[21] A. Kleppe and J. Warmer *Extending OCL to include Actions*, Third International Conference on the Unified Modelling Language (UML 2000), York, UK, October 2000, Lecture Notes in Computer Science vol. 1939, pp. 440-450, 2000.

[22] A. Lakas, G. Blair and A. Chetwynd, *A Formal Approach to the Design of QoS Parameters in Multimedia Systems*, Proceedings of the 4th International Workshop on Quality of Service, Paris, France, March 1996. Internal report number MPG-96-26, Lancaster University, UK.

[23] P. F. Linington, *RM-ODP: The Architecture*, In K. Raymond and E. Armstrong, editors, IFIP TC6 International conference on Open Distributed Processing, pp. 15-33, Brisbane, Australia, February1995. Chapman and Hall.

[24] P. F. Linington, *Options for expressing ODP Enterprise Communities and their Policies by using UML*, Proceedings of the Third International Enterprise Distributed Object Computing Conference, pages 72-82. IEEE, September 1999.

[25] L. Mandel and M. V. Cengarle, *On the Expressive power of the Object Constraint Language OCL*, World Congress on Formal Methods in the Development of Computing Systems, FM99, LNCS 1708, pp. 1999.

[26] F. Pinet and A Lbath, *Semantics of Stereotypes for Type Specification in UML: Theory and Practice,* 20th International Conference on Conceptual Modeling (ER'2001), Lecture Notes in Computer Science vol.2224, Springer-Verlag, pp.339-353, November 2001.

[27] J. Rumbough, J.M. Blaha, W. Permerlani, F. Eddy and W. Lorenson, *Object-Oriented Modelling and Design*, Englewood Cliffs, Prentice Hall, 1991.

[28] K. Raymond, *Reference Model of the Open Distributed Processing (RM-ODP): Introduction The Architecture*, In K. Raymond and E. Armstrong, editors, IFIP TC6 International conference on Open Distributed Processing, pp. 3-14, Brisbane, Australia, February1995. Chapman and Hall.

[29] *Resource Description Framework*, W3C Technology and Society Domain, www.w3.org/RDF

[30] F. Singhoff and I. Demeure, *Spcification et ordonnancement dynamique d'applications multimdias : l'environnement POLKA*, Real time systems (RTS 98), Paris la Dfense Janvier, pp. 101-115, 1998.

[31] M.W.A Steen and J. Derrick, *Enterprise viewpoint Specification*, Computer Standards and Interfaces, Vol. 22, 165-189, 2000.

[32] R. Steinmetz and K. Nahrstedt, *Fundamentals in Multimedia Computing and Communications*, Englewood Cliffs, Prentice-Hall, 1999.

[33] TINA-C, *TINA object definition language*, July 1996. Available from http://www.tinac.com/specifications/documents/odl96_public.pdf.

[34] K.J. Turner, *Using formal Description techniques, an introduction to Estelle, LOTOS and SDL*, Addison-Wiley, 1993.

[35] J. Oldevik and A.-J. Berre, *UML based methodology for distributed systems*, 2nd International Enterprise Distributed Object Computing Conference (EDOC 98) San Diego **what page**

[36] *UML 1.3 Documentation*, Rational Rose Resource Centre, 1999.

[37] J. Wermer and A. Kleppe, *The Object Constraint Language, precise modelling with UML*, Addison-Wesley, 1999.

[38] *World Wide Web Consortium*, www.w3.org.

[39] *Extensible Markup Lanuage*, W3C Architecture Domain, www.w3.org/XML