Computer Science at Kent

Design and Verification of Distributed Multi-media Systems

D.H.Akehurst, B.Bordbar, J.Derrick,
A.G.Waters

Technical Report No. 1-03
January 2003

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent CT2 7NF, UK

Design and Verification of Distributed Multi-media Systems

D.H.Akehurst, B.Bordbar, J.Derrick, A.G.Waters

University of Kent at Canterbury
{ D.H.Akehurst, B.Bordbar, J.Derrick, A.G.Waters }@ukc.ac.uk

Performance analysis of computing systems, in particular distributed computing systems, is a complex process. Analysing the complex flows and interactions between multiple threads over a distributed set of processing nodes is a non-trivial task. The problem is exacerbated by the addition of continuous system functions that are time dependent, such as communication between components in the form of multimedia streams of video and audio data. Quality-of-Service (QoS) specifications define constraints on such communications and describe the required patterns of data transfer. By making use of these specifications as part of the performance analysis process it is possible to add significant confidence to predictions about the correct (required) operation of a distributed system. This paper presents a method for designing distributed multimedia systems, including the specification of QoS, using the ODP framework and UML and describes a technique for verifying the QoS specification against the designed functional behaviour of the system using Timed Automata.

Keywords: ODP, Timed Automata, UML, QoS, Performance, Multi-media

1 Introduction

The problems of designing computing systems are as old as the technology supporting their implementation. As computing systems become larger and more complex, addressing the problems becomes more of an issue. Recent advances in technology have caused new functionality to be expected of modern computer systems. One of these expectations is the ability to provide multi-media facilities, including the transmission and manipulation of data streams, such as audio and video flows. Another expectation is that the processing resources of the system will be distributed across multiple nodes, which in turn are quite possibly located in a geographical disparate fashion.

Recent years have seen a substantial increase in the introduction of devices that support distributed and multimedia communication, into our daily life – an example of which is the multi-media enabled mobile phone. There is a growing need to address the issues of performance as a part of the development and design of systems that make use of such devices.

The telecommunications community has long been investigating the problems of designing distributed systems and has standardised on a number of issues. Some of these standards address the design of distributed systems [1, 2] through the use of a design framework known as Open Distributed Processing (ODP); and some address specifically the issue of specifying stream communication and the definition of the quality at which the computing system is expected to provide its services – commonly known as Quality of Service [3, 4].

A Quality of Service (QoS) specification is the definition of some timeliness, reliability, or availability need of a business, end user, or software entity. From the user's point of view, the quality of the performance of a distributed system is end-to-end, i.e. it is not important if certain parts of the system fail to deliver some performance, as long as the system as a whole provides the expected level of performance. A major challenge of the process of integrating QoS is to specify suitable QoS characteristics for each component of a distributed environment such that the overall system satisfies the expected level of performance. A number of languages have been proposed as methods for defining QoS - CQML [5], SMIL [6], TINA-ODL [7], QTL [8], QuO [9] amongst others – and QoS issues have been incorporated into some implementation frameworks; for example, the latest CORBA 3 standard [10, 11] includes mechanisms for building CORBA systems that support QoS negotiations and stream communication.

The QoS specified for each component of a distributed system often affects the behaviour of the system by imposing restrictions on the functional behaviour of that component. For example, a QoS statement might impose a restriction on the relative time of occurrence of a sequence of events partaking in the behaviour of the component As a result, imposing such restrictions inevitably, influences the overall behaviour of the system. In particular, an over-demanding QoS specification on a component can be beyond the physical limitation of the systems and might result in inconsistency among QoS specification and functional behaviour of the components.

The aim of this paper is to illustrate our approach to designing distributed systems using UML and to demonstrate a technique for analysing the specification of distributed systems that include both QoS and behaviour definitions. Our technique gives feedback to a system designer regarding whether or not a particular configuration of system components with defined behaviour will satisfy the specified QoS constraints.

We first illustrate our approach to the design of distributed systems. Using an example, we indicate a design method and demonstrate use of the languages we adopt for the specification of distributed system structure, behaviour and QoS.

One aspect of our approach to the design of computing systems is to make use of the current common and best practises and tools that support the design of the types of distributed multimedia systems in which we are interested; this approach should result in widely understood specifications and gain us maximum support from existing design tools. Currently, the Unified Modelling Language (UML) [12] is by common practise a clear contender for the design language of choice. However, although having significant community and tool support, it does not provide a means to address some of the issues relevant to distribution and multimedia. To support the design of such systems we look to the ODP design framework standardised by the ITU/ISO bodies. The framework does not prescribe the use of any particular languages; hence where appropriate and possible we make use of the UML, e.g. for the specification of behaviour we use the UML State Diagram [12] notation; where the UML does not provide us with an appropriate language, we adopt other languages that prove to be compatible and would have a familiar 'feel' to a designer experienced in using the UML.

The approach described in this paper, builds on our work from a previous project [13], which address performance prediction from UML system designs and earlier work from this project [14, 15], regarding the UML and specification of QoS. While [15] deals with the static aspects of specification of QoS, the current work aims to address the implication of the specified QoS on dynamic aspects of the system.

The ODP model of the computing system under design embodies the definition of structure, behaviour and QoS of the system. Our analysis technique presents a method to translate designs written using the approach described in this paper into a set of Timed Automata (TA) [16]. The TA corresponding to the QoS specification and the TA corresponding to the functional behaviour are composed together in accordance with the structural specification to produce a TA model of the behaviour and requirements of the system. This composition of parallel TA is analysed using the model checker UPPAAL [17], which provides feedback on the dynamic behaviour of the system. This feedback can subsequently be used to improve the design of the system.

To illustrate the techniques proposed in this paper we introduce part of an example system based on a possible future application of the multi-media technology of mobile phones. Current, state-of-the-art, mobile phones provide facilities for sending text and pictures along with the traditional voice communication expected of a phone. Before long, it is likely that video communication will also be included; in this case, it is likely that audio and video data will be transmitted via different streams and protocols. Given that network communication is not perfect, both streams are likely to be affected by the characteristics know as latency and jitter. When reconstructing the combination of audio and video at the destination endpoint of the stream, some form of synchronisation is likely to be required. An algorithm and design for a possible lip-synchronisation system is given in [18]; we use their example as a basis for our own.

The rest of this paper is organised as follows. Section 2 gives an introduction to the ODP framework. Section 3 elaborates the details of our example system using the ODP framework and our design approach. Section 4 shows the translation from specifications in our design languages into Timed Automata for both the structure, behaviour and QoS of the system. Section 5 demonstrates the analysis technique incorporating the use of the UPPAAL model checker and illustrates how results can be fed back to the designer. The paper concludes in section 6 with a summary and discussion of the presented, future and related work.

2 Open Distributed Processing

The ODP framework proposes a multi-paradigm specification approach for the design of distributed systems by identifying five separations of concern and addresses the design of the system from each. Different languages may be used for each separation of concern providing the benefit that the relative strengths of different specification languages can be exploited. In ODP terminology these five separations of concern are named viewpoints.

The reference model for ODP [1] defines five viewpoints: *enterprise*, *information*, *computational*, *engineering* and *technology*; further information regarding both the reference model and its approach to

using viewpoints can be found in [1, 19, 20]. The design approach and analysis technique proposed in this paper is explored within the scope of the computational viewpoint, although it may well be applicable to the others, in particular the engineering viewpoint.

The computational viewpoint is concerned with the identification of distributable components (objects) and their interaction points (interfaces). The viewpoint addresses: the specification of the behaviour of identified objects; the specification of the signatures of the interfaces through which they interact; the specification of templates from which such components can be instantiated; and the specification of any constraints under which the objects must operate. The mechanism by which distributed communication is achieved is not addressed in this viewpoint (that is part of the purpose of the engineering viewpoint), i.e. *distribution transparency* is assumed.

The following section provides an example computational specification with the description of a method for creating such design.

3 Computational Specification of the Lip Synchronisation System

In this paper we demonstrate our verification technique on a computational viewpoint specification. The example used to illustrate the technique is described in the following sections. We start with a snapshot of the system, which gives an indication of the primary distributable components composing the system and the interfaces required to connect them. From the snapshot we identify and specify the computational object templates and interface signatures of the system. For each computational object we subsequently provide a behaviour specification. Finally we specify environment contracts for each computational object in the form of some QoS constraints.

3.1 System Snapshot

An aspect of a computational viewpoint specification is the decomposition of the system into distributable *objects* that interact at *interface*s. A computational object, which may be a composition of two or more other objects, is a unit of distribution and management that encapsulates behaviour [1]. In particular, computational objects are not instances of classes, as is the case in Object Oriented (OO) languages [20]. To avoid confusion with the word object, which is a 'reserved' word in the UML, we shall use the term *Computational Object*.

Figure 1 depicts a computational viewpoint snapshot of our example mobile videophone, including the lip synchronisation component. Circles depict computational objects; there are five of these – *camera*, *videoWindow*, *microphone*, *speaker* and *syncController*. The computational objects *videoCamera* and *microphone* are part of the transmitter device and capture the video frames and audio packets at the source of transmission. These packets and frames are transferred via two separate (video and audio) channels to the receiver device's computational objects *videoWindow* and *speaker*. Binding objects are distinguished from computational objects by illustrating them as elongated circles.

Interfaces are illustrated using 'T' shapes, attached to a circle to indicate that the object (depicted by the circle) offers that particular interface. The role of the interface (producer/consumer, initiator/responder or client/server) is indicated by the direction and style of an arrow placed near the interface. Bound interfaces are either connected via an irregularly dashed line (e.g. *r_vidCtrl* and *i_vidCtrl*) or placed head to head (all other bound interfaces in this snapshot – *transVideo*, *recVideo*, *transAudio*, *recAudio*, *vidReady*, *audReady* and *appControl*).

The identification policy for objects and interfaces is similar to the approach used in UML object diagrams, computational objects and interfaces are identified by either or both of an 'instance name' and a 'template name' separated by a colon and underlined. Where bound interfaces are close together we omit naming both interfaces separately and distinguish between them using their role.

In this snapshot the two bindings (for audio and video) are labelled with only the template name (see following section) other objects and interfaces are labelled with only an instance name.

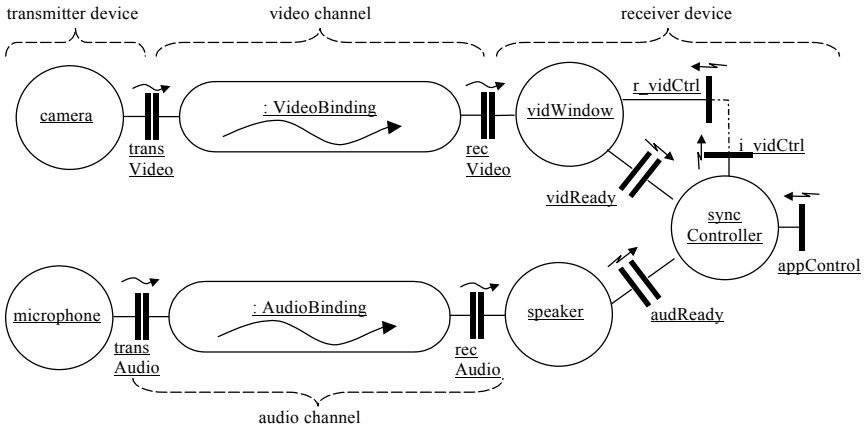


Figure 1 Computational Viewpoint snapshot illustrating the Lip Synchronisation system

The transmitter device's *camera* emits video frames to the *videoBinding* across the bound *transVideo* interfaces. The receiver device's v*idWindow* receives the video frames from the *VideoBinding* at the bound *recVideo* interfaces. On arrival of each video frame a signal is output to the *syncController* at the bound *vidReady* interfaces. Similarly the audio packets are emitted from the *microphone* computational object to the *AudioBinding* at the *transAudio* interfaces; they arrive across the binding at the speaker computational object and *recAudio* interfaces, where signals are emitted to the *syncController* at the *audReady* interfaces.

Transmission of the audio and video data via different channels applies a different and variable time delay to the media streams. As a result the computational object *syncController* is used to adjust the playback rate of the media streams to produce synchronised presentation. The *syncController* indicates when to display each video frame by emitting a signal at its *i_vidCtrl* interface, which is received by the *vidWindow* at *r_vidCtrl*. Finally, the s*yncController* has an interface *appControl* that is used to reset the synchronisation algorithm.

3.2 Template and Signature Specifications

The snapshot discussed in the previous subsection indicates the kinds of component needed in order to build the system. The next step is to fully specify those components in order to obtain reusable and detailed definitions of the aggregated parts of the system. From a computational viewpoint, the necessary specifications include the definition of computational object templates, interface signatures, and the relationships between them.

Over the last decade considerable interest has been shown in the Unified Modelling Language [12] for the design and specification of object-based systems. Since the UML provides a rich set of notations for capturing various aspects of computing systems, the ODP community has shown extensive interest in using parts of the UML to specify various parts of ODP designs [21-24].

However, the ODP concept of an object is not entirely compatible with the UML (and other) Object Oriented (OO) concept of an object. There are two subtle differences:

1) An ODP class is a set of entities that satisfy a type (i.e. a specification of how to *classify* objects), where as a UML class is the specification of how to *construct* an object; and
2) An interface supported by an ODP object provides a communications port, whereas a UML interface is a type classifier.

In UML the *class* tends to be the focus of modelling, an *object* simply being an instance of a class. In ODP, the object itself is the focus of modelling; object instantiation is through a defined *object template*, rather than a class. The word *class* in the ODP context refers to the set of all entities that satisfy some *type*. So in ODP we can talk of a class of objects of type X, a class of interfaces of type Y or a class of templates of type Z. A *type* in ODP refers to a *predicate*, a set of conditions to classify an element of the system and which can be evaluated for all elements.

An identifying feature of classes in ODP is that an ODP class, being a set, can be empty, i.e. nothing satisfies a given type, though it may later have members. On the other hand, templates are patterns of feature. In particular, an interface signature (template) defines the type of the interface and the interactions that may occur across that interface. For each interaction, the interface template defines the name and type of the interaction, the types parameters, the directionality and the exceptions raised. As a result, the normal OO concept of class relates more closely to the RM-ODP construct for a template.

The relationship between objects and interfaces in the UML world is one of *realization*. A UML interface defines a particular set of features; to realize an interface, an object (defined by a UML class definition) implements the defined set of features. I.e. with respect to an interface, the features are abstract definitions, that are only 'made real' by an object.

Within the ODP, interfaces are more of a first class entity; ODP objects *offer* a number of interfaces, through which interactions, both incoming and outgoing, occur. The same interface signature may be instantiated and offered by an object multiple times – offering the same set of interactions to multiple different peers. A particular point to note is that both input and output communications require an interface in the ODP world – unlike the UML, which only facilitates the specification of incoming communications, there is no means to explicitly specify what outgoing operations an object may call.

A consequence of these differences is that we cannot use UML class diagrams "as is" to model the structure of distributed systems within our approach. The semantics of a UML class and its relationships are not wholly compatible with the ODP semantics of templates. However, given that the UML allows us to 'stereotype' its design concepts, enabling us to effectively define our own concepts, we do so and reuse the design notation of UML class diagrams as a notation for those concepts.

The UML concept of a class has a similarity to the ODP notion of a template (or signature as it is named for interfaces) we define stereotypes of the UML class to define the ODP concepts of: computational viewpoint object templates; stream, operational and signal binding objects; reactive objects; and stream, operational and signal interface signatures. This gives us a language and notation suitable for defining the computational viewpoint of an ODP system, which is (hopefully) familiar to UML designers; easily used; and provided with tool support from many standard UML tools.

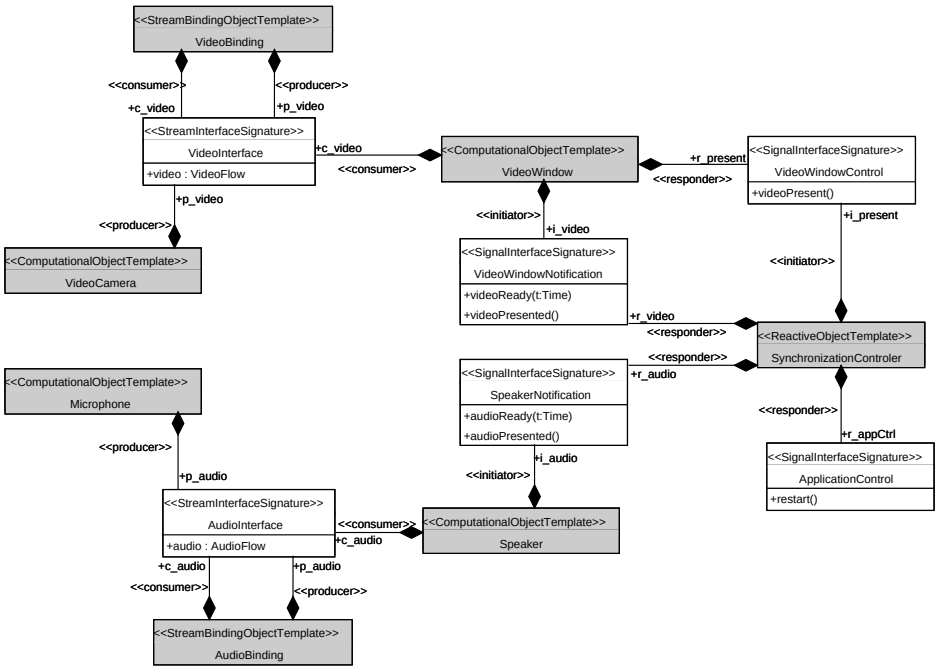


Figure 2 Computational Template Diagram for the Lip Synchronisation system

Figure 2 defines a template diagram for the computational snapshot shown in Figure 1. Both object template and interface signatures are depicted using the notation for UML classes, distinguished using stereotype labels. To aid the distinction, computational object and binding object templates are shaded, whereas interface signatures are not. The stereotype of interface signatures also distinguishes between operations, stream and signal signatures.

The relationship between an object template and the interfaces that its instances may offer is defined using stereotyped UML associations. The stereotype of the association defines the role in which the object may offer instances of the interface signature; the association end name gives a navigation name for the object to refer to the interface. Each interface instance may be offered by only one object; hence the object end of the association is defined to be an aggregation (using a black diamond).

3.3 Behaviour

After defining the object templates and the interfaces they may support, it is necessary to define the behaviour of the objects and the interactions that occur across the interfaces. This subsection firstly describes our adopted approach to specifying behaviour and subsequently illustrates the techniques by defining the behaviour of the three receiver device computational object templates – VideoWindow, Speaker and SynchronizationController.

As stated in the introduction, one of our requirements is to use common design practices; following this directive we look to the UML for a notation that enables the specification of state-based behaviour. The UML defines a particular variant of state and transition based behaviour, based on (a subset of) the formalism of *Statecharts* [25, 26], and renamed *State Diagrams* within the context of the UML. A state diagram represents the behaviour of entities capable of dynamic behaviour by specifying its response to the receipt of event instances. A state diagram consists of *states* and *transitions*.

A state is a condition during the life of an object or an interaction, during which it satisfies some condition, performs some action, or waits for some event. A state is normally depicted as a rectangle with rounded corners, although special types of state are depicted in other ways.

A transition is a relationship between two states indicating that an instance in the first state will enter the second state and perform specific *action*s when a specified *event* occurs provided that certain *guard* conditions are satisfied. A transition is shown as a solid line originating from the *source* state and terminated by an arrow on the *target* state; a transition is typically labelled with a string that has the following general format:

```
<event-signature> '[' <guard-condition> ']' '/' <comma-separated-action-expressions>
```

Where *event-signature* describes an event with its arguments, *guard-condition* is a boolean expression written in terms of parameters of the triggering event and attributes of the object whose behaviour is described by the state machine. The *action-expressions* are executed if and when the transition fires (i.e. the source state is active, the event occurs and the guard evaluates to true). Actions are expressions that either:

1) Alter the local (attribute encoded) state of the object;
2) Instantiate interfaces to be offered by the object; or
3) Cause interactions at a specified interface.

An action must be executed entirely before any following actions are considered – i.e. actions are considered atomic.

State diagrams also allow hierarchical nesting of states; this enables complex behaviour to be specified in a concise manner. Sub-states are either single state diagrams in which contained states are 'or'-states in that one or other is active; or alternatively sub-states can be concurrent where by both sub-states are active and generally further refined to a sub state-diagram (e.g. Figure 6).

The examples in this paper make use of four special types of state (known in the UML as pseudo states). These are: the *initial state*, where the behaviour starts, depicted as a filled circle with a single outgoing transition; the *final state*, where the behaviour terminates, depicted as a filled circle inside a hollow circle; the *choice state*, causing dynamic evaluation of guards to determine the behavioural path, depicted as a hollow circle; and the *junction state*, which enables multiple transitions to be chained or merged together. Pseudo states are not assumed to be stable – i.e. the state machine should not 'wait' for an event to occur within a pseudo state.

Speaker

Figure 3 depicts the specification of our example speaker object's behaviour. The *speaker* object (of Figure 1) emits an *audioReady* signal at its initiator interface *i_audio*, whenever an audio packet is received at the consumer interface *c_audio*; the parameter of which contains the audio packet's timestamp – *c_audio.timestamp* – recording when the packet was created at source. (The name of an interface can be used to access data associated with the last event occurring at that interface.) Audio packets are processed immediately and when the presentation of the packet has completed an *audioPresented* signal is emitted at *i_audio*.

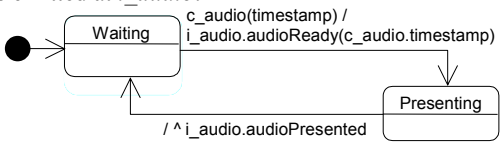


Figure 3 Statechart specification of the Speaker's behaviour

VideoWindow

Similarly, Figure 4 shows the specification of the behaviour for VideoWindow objects. Video frames are received at consumer interface *c_video*, the object emits a *videoReady* signal from initiator interface *i_video* whenever a frame arrives; the associated timestamp indicates the time at which the frame was captured (at source). Frames are not presented immediately; they are presented when the object receives a *videoPresent* signal at the *r_present* responder interface. Once the frame has been presented a *videoPresented* signal is emitted at *i_video*.

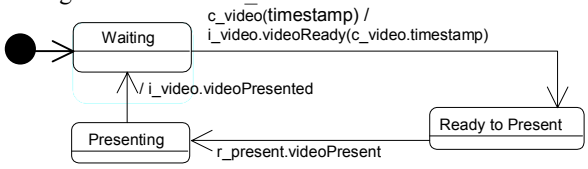


Figure 4 Statechart specification of the Video Window's behaviour

Synchronisation Controller

The Synchronisation Controller objects have a complex specification. For a full description of the algorithm and the way in which it works refer to [18], it is not the purpose of this paper to describe a synchronisation algorithm, we simply use it as an example. An overview description of the behaviour is given below, along with a state diagram (Figure 6) that specifies the behaviour.

The behaviour of the controller consists of two concurrently executing parts; the audio part and the video part. The only communication between these two parts is the value of the variable 'audioDelay', which is written to by the audio part and read by the video part. Regardless of the current state of the controller, a restart signal completely resets the controller and its internal computations.

Within each of these two concurrent sub-states, in addition to transitions that react to and cause signals, there are some internal calculations requiring local data (such as AudThisDelay, ComparedDelay, etc) and these calculations make use of constant values (i.e. VIDTHEORATE, DISPVIDDELAY, etc). These variables and constants can be declared as part of the Object Template specification and referred to within the state diagram. The declarations can be specified as shown in Figure 5. (For a full explanation of the use of these variables and values the reader is referred to the original text from which the example was taken [18].) In addition to these constants and variables, there is an operation to return the current system time.

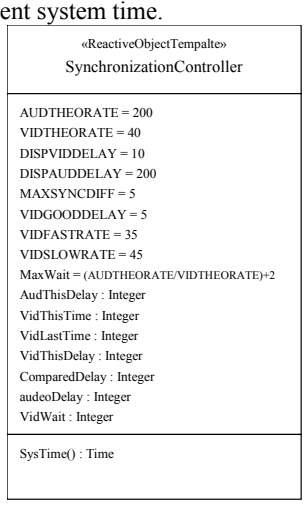


Figure 5 Synchronisation Controller Variables, Constants and Operations

The audio part of the controller reacts to every audioReady signal by calculating the delay for the appropriate audio packet (given by the current time minus the time the audio packet was captured). Following this calculation the audio part of the controller waits until an expected number of video frames have passed and then indicates that the next audio packet is missing, by setting audioDelay to 0. Normally, however a new audio packet will arrive before this happens (and consequently a new audioReady signal) and the new audioDelay value is calculated.

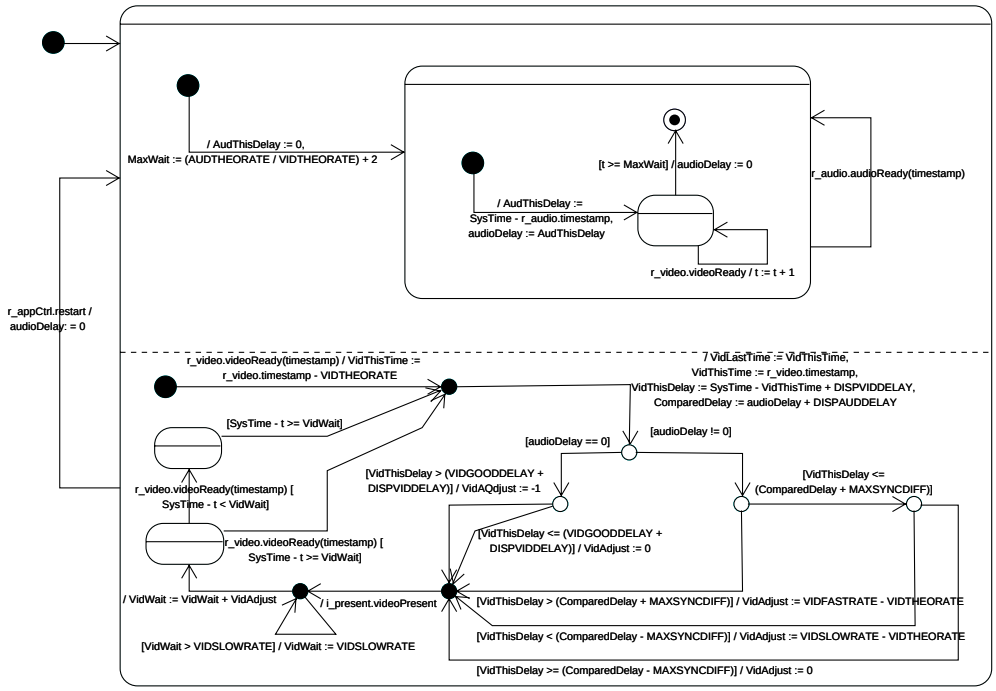


Figure 6 Statechart specification of the Synchronisation Controller's behaviour

The video part of the controller starts by waiting for the first videoReady signal (i.e. the first frame) and sets the capture time of a fictional previous frame to be the time of the current frame minus the ideal inter frame gap. After the initial frame, all further frames are handled in the same (following) manner; the inter frame gap is computed (between the current frame and previous), the videoDelay of the current frame is calculated (by comparing the current time with the capture time of the frame) and the difference (ComparedDelay) between the current audioDelay and the current videoDelay is computed. These values are used to determine whether to speed up or slow down the video display rate:-

- If the audio is missing (audioDelay = 0) we speed up the display rate if the video transmission delay is above a threshold.
- If the video is late, then speed up the display rate.
- If the video is early, then slow down the display rate.
- Otherwise do not adjust the display rate.

After calculating this rate adjustment, display the current frame and then wait for either the next video ready signal (next frame to arrive) or for the calculated wait time, which ever takes longest. The wait time is calculated by using the previous inter frame gap, adjusted by the rate adjustment previously calculated.

3.4 Environment Contracts – QoS Specification

The previous subsections have defined the structure, templates and functional behaviour of the system. Now we address the specification of non-functional aspects of the system by defining some QoS constraints.

The ODP standard defines the concept of an environment contract. This is a contract between an object and its environment, i.e. all other object with which it interacts. As interactions occur across interfaces, environment contacts for an object generally involve one or more interfaces. A QoS constraint is one such example of an environment contract. Such a constraint involves two parts:

1) Requirements of the object by the environment, known as obligations; and
2) Requirements of the environment by the object, known as expectations.

The relationship between these two parts states that provided the expectations are met (by the environment) the obligations will be met (by the object).

There is currently no clear contender for a most commonly used (de facto) QoS language. Many have been proposed [5-9]; one that we have found to be most suited to our design approach, partly due to its association with the Object Constraint Language (OCL) [27] and UML, is the Component Quality

Modelling Language (CQML) [28, 29]. CQML is a lexical language for QoS specification and has been developed to explicitly include as many features as possible [5]. We have found the language to be expressive, very useable and easily integrated with our other UML based languages within the ODP framework.

There are many possible QoS characteristics that could be constrained, [30] lists those identified by the ITU. For the purpose of this paper we look at three stream and time related characteristics – latency, anchored jitter and throughput. *Latency* is the amount of time between two events (e.g. time between sending a frame and receiving it); throughput is the rate of occurrence of events (e.g. the rate of flow of frames); and *anchored jitter* is a variation in nominal throughput.

CQML facilitates the definition of *quality characteristic*s such as latency, throughput and anchored jitter, in terms of the history of events at a particular interface (we do not provide the definition of these characteristics in this paper). The semantics define that each interface instance contains a historical sequence of events. This is essentially provision of the 'Event Notification Function', defined by the RM-ODP [1], which requires event histories to be made available.

Constraints regarding particular characteristics are formed in CQML by specifying *quality statement*s, these are grouped to form QoS specifications on particular objects or object templates as *QoS Profile*s. A QoS profile includes statements for both e*xpectations* and *obligations*; each *expectation* or *obligation* is an expression referring to one or more quality statements. The quality statements enable reuse of QoS specifications across multiple QoS profiles.

A quality statement contains the conjunction of a number of sub expressions that constrain a variety of quality characteristics. Each quality characteristic is defined by an OCL expression that (in the case of latency, anchored jitter and throughput) references the associated event histories. To enable quality characteristics to be generalised and reused, they can be defined with specific parameters.

Given a set of pre-defined quality characteristics (throughput, anchoredJitter and latency) the QoS specifications associated with the Template diagram of Figure 2 (defining templates for the system illustrated in Figure 1) can be defined and explained as follows. We do not include all of the QoS specifications for the system, but show only those needed by the rest of the paper; i.e. QoS specifications for the *VideoBinding*, *VideoWindow*, *Speaker* and *SyncController*.

VideoBinding

The VideoBinding from camera to video window is specified to provide a through frame rate of no less than 25 fps with a latency of between 40 and 60 milliseconds (ms) so long as it receives an input frame rate of no less than 25 fps. This is expressed in CQML as follows:

```
QosProfile for VideoBinding {
  exp: quality { throughput(1000, c_video.video) >= 25; };
  obl: quality {
        throughput(1000, p_video.video) >= 25;
        latency(c_video.video, p_video.video).maximum = 60;
        latency(c_video.video, p_video.video).minimum = 40;
       };
}
```

The above *QoS Profile*, defined for the *VideoBinding* template, defines one *expectation*, that there should be at least 25 events received every second (1000 ms) at the '*video*' *VidowFlow* part of the consumer interface *c_video*. It also defines that the binding is *obliged* to provide at least 25 frames every second (fps) from the *VideoFlow* (named '*video*') part of the *VideoInterface* signature (*p_video*) supported by the binding in the role of a *producer*. The particular *VideoFlows* on which the constraints are placed is navigated to using the association end names of the associations relating object templates to interface signatures. Additionally there are constraints between consumer and producer *VideoFlows* that specify the maximum and minimum latency that should occur for a frame passing through the binding.

VideoWindow

The Video Window expects an input video stream at a rate of no less than 25 fps; if it gets this it will present frames at a rate of no less than 25 fps. The *videoReady* signal will be produced immediately that a frame is received and the frame will be presented no later than 10 ms after being received. We also assert that the *VideoWindow* will take no more than 5ms to present the frame (once instructed to do so). In addition we include an obligation of meeting a ±10ms jitter on the presentation rate. In CQML, these constraints are specified as follows:

```
QosProfile for VideoWindow {
  exp: quality { throughput(1000, c_video.video) >= 25; };
  obl: quality {
        throughput(1000, i_video.videoPresented) >= 25;
        latency(c_video.video, i_video.videoReady) = 0;
        latency(c_video.video, i_video.videoPresented) <= 10;
        latency(r_present.videoPresent, i_video.videoPresented) <= 5;
```

```
        anchoredJitter(i_video.videoPresented) < 10;
      };
}
```

Speaker

The Speaker expects an input audio flow at a rate of no less than 5 packets per second (pps). The presentation rate of the packets will be at the same rate and each packet will be presented within no more than 200 ms of receipt of the packet. The *audioReady* signal will be emitted immediately upon receipt of audio packets. Defined in CQML, these constraints are as follows:

```
QosProfile for Speaker {
  exp: quality { throughput(1000, c_audio.audio) >= 5; };
  obl: quality {
        throughput(1000, i_audio.audioPresented) >= 5;
        latency(c_audio.audio, i_audio.audioReady) = 0;
        latency(c_audio.audio, i_audio.audioPresented) <= 200;
      };
}
```

SyncornizationController

There are no QoS constraints defined on the Synchronisation Controller. This can be illustrated in CQML by an empty QoS profile:

```
QosProfile for SynchronisationController {
  exp:;
  obl:;
}
```

4 Translation to Timed Automata

To verify the QoS specifications against the functional behaviour of a particular configuration of the system components we create a network of parallel timed automata. The first step towards this is to convert the CQML QoS specifications into TA (as described below) and to convert the state diagram specifications of the functional behaviour into TA. State diagrams are mapped to hierarchical timed automata [31] which can be flattened for input to UPPAAL using the method described in [31]. This gives a network of TA that model the whole system, plus the environment in which the system is supposed to execute.

An automaton of the functional behaviour for an object that interacts with its environment could not be executed or checked in isolation; the automaton would stop as soon as any input interaction point was reached. However, the specification of QoS for an object defines the expected behaviour of the environment – i.e. it defines the characteristics of events from the environment. An automaton that models these ‘expectation’ characteristics can be placed in parallel with the functional behaviour, thus providing the required input signals. The definition of QoS states that obligations will be met if expectations are met – given that the expectations are defining the inputs it can be assumed that they are met; hence to verify the whole QoS specification it remains to determine whether or not the output signals meet the specified obligations. This is achieved by defining additional ‘monitoring’ automata that enter defined states if the obligations are not met.

The analysis platform (UPPAAL) on to which we target our TA models uses a template and instantiation model (similar to the concepts defined for ODP). The details (states, transitions, guards, clocks etc.) of individual automata are specified as a parameterised template. Networks of automata are specified by defining instances of templates, which provide values for the parameters. These ‘instance’ automata are synchronised using globally defined ‘channels’; each end of a channel corresponds to a synchronised event occurring in each of two automata.

We do not currently have a generic mechanism for translating CQML *quality characteristics* into a TA template; however, for each of the three characteristics we are handling (latency, anchored jitter and throughput) it is possible to construct (by hand) a TA template that models the characteristic. The parameter values to instantiate these (QoS characteristic) templates (for a particular system) are taken from the quality statements that form part of the CQML QoS profile for each object.

The TA for an obligation must be purely passive and simply monitor the occurrence of events, and report whether or not the specification has been met. To ensure the progression of time throughout our automatons, TA modelling expectation characteristics force events to occur at (or between) specified times. Hence, the automaton template for a characteristic constrained in an expectation is different to the automaton template for the same characteristic but constrained within an obligation.

4.1 Throughput and Anchored Jitter

The characteristic of throughput is easily modelled as a continuous source of events; however, due to the properties of anchored jitter, it is unnecessary to separately model throughput and we can define

a single TA that models both. Anchored jitter is defined as a variation of throughput; the variation does not change the overall throughput rate (this would be non-anchored jitter), it simply means that given the expected time for arrival of an event, the event might arrive one or other side of this time. The expected arrival time of following events are not affected by the actual arrival time of events (as is the case with non-anchored jitter). Thus, the specification of anchored jitter is tied to the specification of a particular throughput (which gives the expected arrival times) and hence it is more efficient to model the two characteristics as a single automaton template.

A QoS expectation relating to throughput and anchored jitter can be encoded as the TA illustrated in Figure 7 (adapted from the automaton described in [32]). The automaton offers output events at a regular rate with an inter-event gap of 'RATE' (inverse of throughput) with an anchored jitter of between 'MAX' and 'MIN' above and below the value 'RATE'. The variable 'data' models a parameter value carried by the output event, for example a frame count, or as in the case of the lip sync example can hold a timestamp for the time the event (frame or packet) is created.

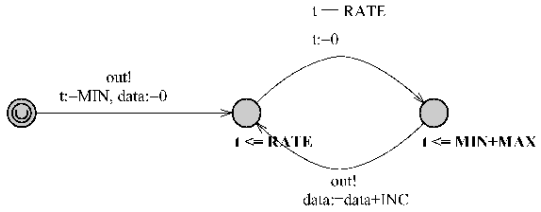


Figure 7 Expectation Anchored Jitter and Throughput

A pair of CQML quality statements defining the throughput and anchored jitter at a particular interface map to the parameter values of this automaton as follows:

For the quality statements:

```
quality {
  throughput >= X;
  anchoredJitter <= Y;
}
```

The parameters RATE, MIN and MAX would be defined as:

```
RATE = 1 / X
MIN = MAX = Y
```

Note: The automaton models throughput using a fixed (through jittery) inter-arrival rate. As such it does not truly model the throughput characteristic used in the QoS specification, which specifies a constraint on the total number of frames arriving within a specified time (i.e. 1000 milliseconds). Strictly speaking it is modelling the quality statement:

```
quality {
  minimum_inter_arrival_time = 1/X;
  anchoredJitter <= Y;
}
```

Similarly, the TA shown in Figure 8 assures a QoS obligation involving throughput and jitter. Note the important difference between the obliged and expected versions of this automaton. The obliged version is acting as an observer of offered outputs and does not force an output to occur.

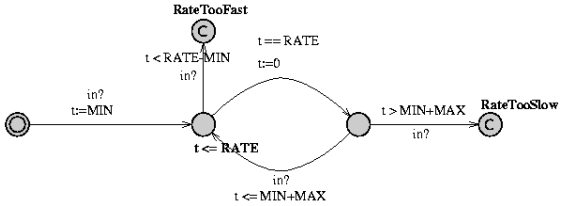


Figure 8 Obligation Anchored Jitter and Throughput

The automaton will deadlock in state 'RateTooFast' if an output is offered too early; i.e. before time 'RATE-MIN' after the last output. Similarly it will deadlock in state 'RateTooSlow' if an output is offered too late; i.e. after time 'RATE+MAX' after the last output.

4.2 Latency

The QoS characteristic latency is defined as "the time that elapses between a *stimulus* and the *response* to it", i.e. latency is the relationship between the time one event occurs and the time that an

associated second event occurs. To model this using timed automaton, it is necessary to be able to record the entry time for every stimulus event that is passing through the component by a clock. We can use each such clock to measure the time between stimulus and response. Since, our model of TA can only accept finite number of clocks, we must calculate how many are needed. Given only a latency specification, it is not possible to determine how many clocks are needed (i.e. how many events to track at any one time) and it would seem that a language allowing only a fixed number of clocks would not enable us to model this characteristic. However, no event will be generated (and need to be tracked) if the rate of generation of stimulus events is zero, i.e. without a throughput there is no need to check for latency. If we have a defined throughput, or at least a defined minimum inter arrival time, it is possible to calculate how many clocks are required.

We calculating this value (the maximum number of events to track), SIZE as follows:

```
SIZE = latency * minimum inter arrival time
```

To model this as a Timed Automaton, we use a circular buffer of SIZE number of clocks to record the stimulus event times. The next available clock is reset when a stimulus event occurs and the time value of the clock is compared with the required latency when the corresponding response event occurs. If an attempt is made to reuse a clock before it has been checked (by response event occurrence) then essentially the buffer is too small. However, assuming that events do not arrive quicker than the minimum arrival rate, if the latency we are checking against is met by every event, the above calculation of 'SIZE' ensures that the buffer is not too small; hence, buffer overflow indicates that latency has not been met. If, at any point in time the number of stimulus events passes 'SIZE', then the actual latency of some preceding event must have exceeded the value of latency against which we are checking. The automaton template in Figure 9 achieves this; the variable 't' is an array of clocks, 'v' is an array of corresponding (frame or packet) identifiers, 'eid' gives an index into arrays based on the identity of the arriving frame/packet, 'latency' is a constant holding the value of latency to check, 'start' and 'check' are the stimulus and response channels and 'st_id' and 'ch_id' carry the identity of the frame/packet represented by the stimulus and response channels.

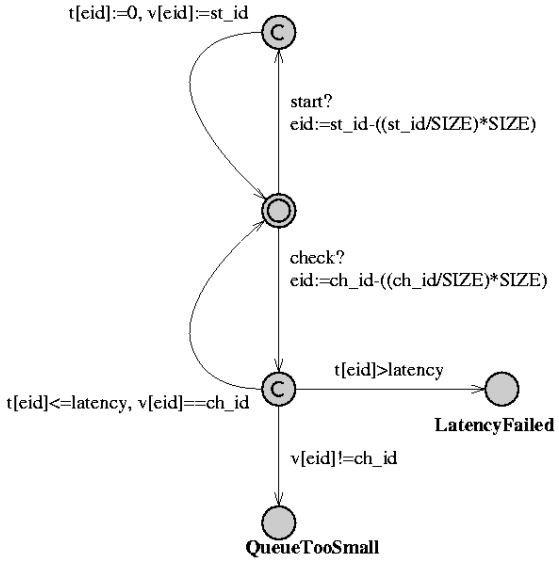


Figure 9 An Automaton Template for Latency Obligation

To enter this Automaton into a model checker such as UPPAAL that does not support arrays of clocks, we must expand the array of clocks explicitly for every value of 'SIZE' that we wish to use. Figure 10 shows an expanded automaton for a buffer size of 5.

5 Performance Analysis

To perform the analysis, i.e. verifying the functional behaviour against the QoS constraints, we produce a set of Timed Automata templates (as described above) for each state diagram specification of the system objects and the QoS expectation and obligation constraints on those objects. These TA templates are instantiated for a particular configuration of object instances (i.e. those specified in a snapshot such as Figure 1). The automata are composed in parallel, synchronising on events; the synchronisation points are deduced from the specified bindings between interfaces described in a

snapshot. QoS automata are instantiated and synchronised in accordance with the interfaces over which the constraints are placed and the values detailing the constraints; this information is taken from the CQML quality statements in each object's QoS profile.

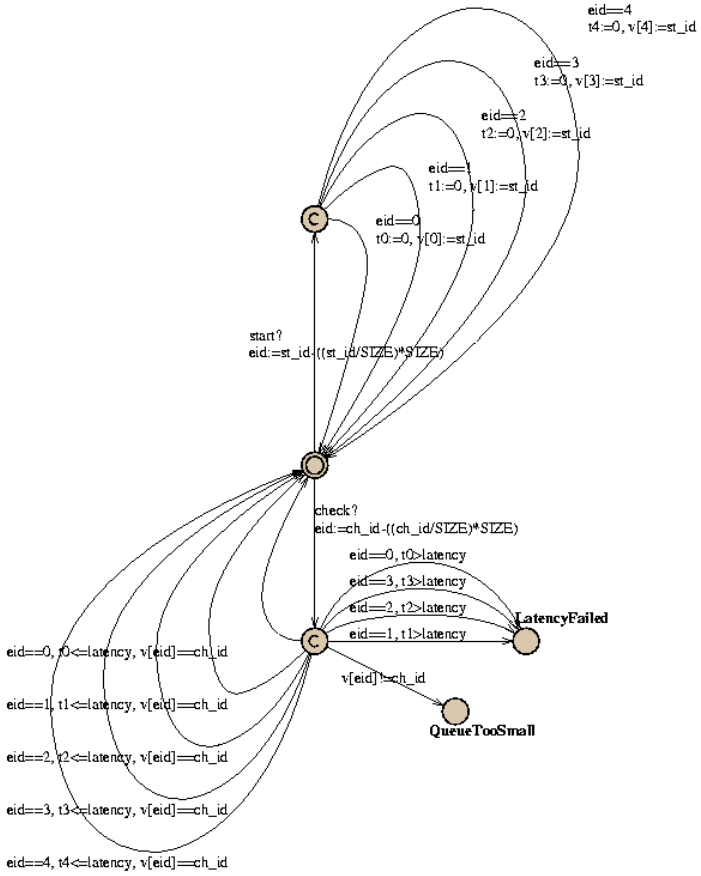


Figure 10 – Expanded Latency Automaton for array size of 5

There are three different sets of automata arising from the translation:

1) Func - Automata modelling the functional behaviour of objects.
2) Obl - Automata modelling the *obligation* QoS statements, which observe the outputs of objects.
3) Exp - Automata modelling the *expectation* QoS statements, which provide inputs to objects from the environment.

The behaviour of the original specification (i.e. the one containing the functional description together with CQML statements of QoS) is now represented by the parallel composition of these three sets of automata:

Func || Obl || Exp

This TA network is entered into the UPPAAL model checking tool [17], which is used to check for deadlock in the system. Formally, there are six situations which could cause deadlock: internal deadlock in an automata from any of the three sets; and deadlock due to a missed synchronisation between a pair of automata each from any pair of the three sets:

a) Deadlock in a functional behaviour automaton (Func): occurs if the functional behaviour is badly designed and causes a deadlock.
b) Deadlock in one of the QoS obligation automata (Obl): indicates that the functional behaviour does not meet the QoS obligation; occurs if the automaton has entered a state that indicates a QoS violation. The QoS obligation automata are designed with specific deadlock states to indicate that the QoS constraints they represent have been violated.
c) Deadlock in one of the QoS expectation automata (Exp): will never occur.
d) Synchronisation deadlock between automata from Func and Obl: will never occur, the Obl automata will enter a specific deadlock state if they can't synchronise on an output event from a Func automata.

e) Synchronisation deadlock between automata from Exp and Func: indicates that the functional behaviour is expecting a different QoS to that provided by the automata from Exp.
f) Synchronisation deadlock between automata from Exp and Obl: will never occur, automata from these sets never synchronise on common events.

Hence, a deadlock is due either to an internal problem of the functional behaviour of a component or due to the functional behaviour not operating in accordance with the obliged or expected QoS.

The position of the deadlock can be fed back to the designer (see example below), indicating which QoS constraint has not been met by the system, or that the functional behaviour has deadlocked. This is illustrated in the following sub-section by analysing our example system.

5.1 Analysing the Example

Figure 11 shows a network of automaton for the example as shown by the 'Simulator View' of UPPAAL in which some of the automata previously described can be identified. The verifier is used to check the model against a query stating that there are no deadlocks for all time:

```
A[] not deadlock
```

To reduce the size of the automata network, we have only included the timed automata for some of the QoS constraints and behaviour of the three computational objects, *videoWindow*, *speaker*, and *synchController*. We are not explicitly modelling the *camera*, *microphone* or binding objects; hence, we must ensure that the same pattern of events is received at the interfaces for Speaker and VideoWindow as would be received if we were explicitly modelling them. This is achieved by adding an additional QoS expectation to represent the jittery arrival of video and audio packets, as would be the case if the variable latency of the bindings had had its affect.

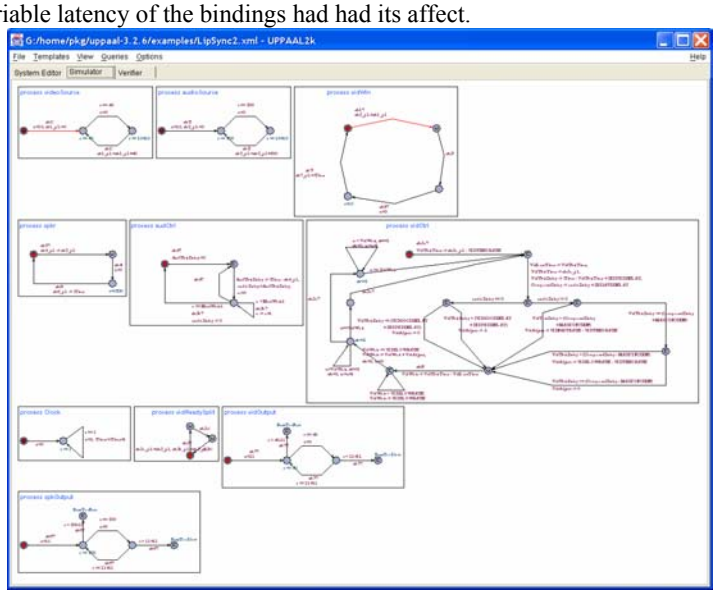


Figure 11 Instantiation of Timed Automata

In general, suppose that the input inter-arrival rate for at an interface I1 of a non-lossy binding is a constant T and the latency between I1 and an output interface I2 of the binding is in the interval [a , b]. Then the binding can be considered to have a fixed latency of (a+b)/2 and the output inter-arrival rate is T ± (b –a)/2.

Applied to our example, the video binding is specified to introduce a latency of between 40 and 60 ms and assumes a constant rate input of packets. Under these conditions, the output of the binding will be a constant rate output with an anchored jitter of ±10ms.

Thus the video stream (as input to the videoWindow) is expected at 25 fps with a jitter of ±10ms, and the audio stream (as input to the speaker) is expected at 5 pps also with a jitter of ±10ms.

The analysis process checks that the obliged QoS of the system is met; i.e. checking the rate and jitter at which the video and audio is actually presented – video presented at >= 25 fps with jitter < 10ms, audio presented at >= 5 pps with jitter < 10ms.

The UPPAAL verifier indicates that there is a deadlock in the network of automata illustrated with the trace of events as described below:

1) The first video frame arrives at the video window, i.e. at a time 50ms after being generated.
2) The videoReady and videoPresent signals both happen with no (significant) time delay, also at time 50.
3) The video window completes presentation with no delay and the videoPresented signal is output immediately, i.e. at time 50.
4) The second frame arrives late at the video window, i.e. at a time 59ms after being generated, i.e. 49ms after the first was received and 99ms after the first was generated.
5) The videoReady and videoPresent signals occur immediately, i.e. at time 99 (relative to first frame generation).
6) The video window takes anything longer than 1ms to present the frame and the videoPresented signal is output at a time > 100ms (relative to first frame generation). This is >50ms since the last videoPresented signal, and does not meet the Obliged Jitter QoS .

The video window is specified to take *at most* 5ms to present a frame. Thus if this varies between 0ms and 5ms, on top of the expected 10ms jitter as input, it follows that the output will vary by more than 10ms. The situation could be aggravated further if the synchronisation algorithm delays the videoPresent signal by (the possible) 5ms within this scenario.

5.2 Feedback of Results

A common criticism of model checking is that the output of model checking tools is a trace in the formal language of the model checker. For example, the trace output from the above example defines a sequence of events and transitions on TA with the time that they occurs or were taken. A trace in this form (Table 1) is of little use a designer who is not familiar with the formal language; the designer did not specify the system in TA. To overcome this we must be able to give feedback to the designer using the language in which he has designed the system.

Table 1 shows a trace of transitions and the time at which the transitions are taken for the situation leading to deadlock described in the previous sub-section. Synchronised transitions are shown as a pair in the same row of the table. A two-part label identifies the transitions. The first part is the instance name of the TA template in which the transition is defined; the second part, in UPPAAL is a number identifying a particular transition, we have replaced this with the name of either ‘internal’ if there is no synchronisation, or the parameter name (specified in the template) for the channel on which the synchronisation is defined.

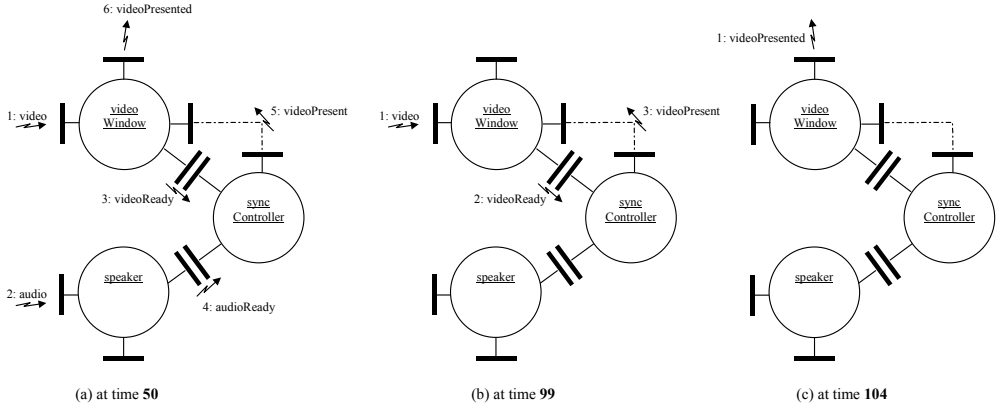
Time	Transition(s)
50	(videoSource.out, vidWin.videoIn)
50	(audioSource.out, speaker.audioIn)
50	(vidWin.videoReady, split.in)
50	(speaker.audioReady, synchController_Audio.audioReady)
50	(synchController.*internal*)
50	(split.out1, synchController_Video.vidReady)
50	(synchController_Video.*internal*)
50	(synchController_Video.*internal*)
50	(synchController_Video.*internal*)
50	(synchController_Video.vidPresent, vidWin.videoPresent)
50	(synchController_Video.*internal*)
50	(split.out2, synchController_Audio.vidReady)
50	(vidWin.videoPresented, vidOutput.in)
50	(synchController_Video.*internal*)
51	(synchController_Video.*internal*)
...	"
78	(synchController_Video.*internal*)
79	(vidOutput.*internal*)
79	(vidSource.*internal*)
79	(synchController_Video.*internal*)
80	(synchController_Video.*internal*)
81	(synchController_Video.*internal*)
82	(synchController_Video.*internal*)
83	(synchController_Video.*internal*)
84	(synchController_Video.*internal*)
84	(synchController_Video.*internal*)
99	(videoSource.out, vidWin.videoIn)
99	(vidWin.videoReady, split.in)

99	(split.out1, synchController_Video.vidReady)
99	(synchController_Video.*internal*)
99	(synchController_Video.*internal*)
99	(synchController_Video.*internal*)
99	(synchController_Video.vidPresent, vidWin.videoPresent)
99	(synchController_Video.*internal*)
99	(split.out2, synchController_Audio.vidReady)
100	(synchController_Video.*internal*)
101	(synchController_Video.*internal*)
102	(synchController_Video.*internal*)
103	(synchController_Video.*internal*)
104	(vidWin.videoPresented, vidOutput.in)
deadlock	***in state* vidOutput.RateTooSlow**

Table 1 UPPAAL Trace of transitions leading to deadlock

The ODP computational viewpoint semantics defines inter object communication to be in the form of signals. Our translation approach maps communication signals between objects to synchronisation events in a TA model and bindings between interfaces map to the synchronisation channels in UPPAAL TA networks; thus the trace of events given by the UPPAAL model checker can be transformed back into a trace of inter-object communications (across interfaces) within the domain of the original (designers) language of the computational viewpoint. The original snapshot diagram that defines the configuration of system components, and consequently the network of automata, can be used as a vehicle to illustrate back to the designer the problematic trace.

Table 1 shows some of the transitions highlighted by a darker background. These are the synchronised transitions that map to communication between objects, the other transitions are internal to an object (as indicated) or relate to internal communication between the audio and video parts of the synchronisation controller. These shaded transitions can be illustrated by a series of computational viewpoint snapshots, which we show in Figure 12.



(a) at time **50** (b) at time **99** (c) at time **104**

Figure 12 – Series of snapshots leading to deadlock/invalid QoS

After analysing this scenario, the designer can determine a solution to the problem and alter the system design in order to remedy the problem. For instance, possible options to solve the problem detected in our example system are as follows:

1) Allow more jitter on the output of the video window,
2) Fix the time taken to present a frame – may not be possible if it is a hardware constraint,
3) Add a buffer on the video stream to reduce the input jitter.

In our example we implement options 2 and 3. We specify a fixed time for video frames to be presented, i.e. specify a fixed latency between signals videoPresent and videoPresented; and state that the anchored jitter on the output rate must be less than 15ms rather than 10 ms. These changes ensure that the jitter on frames being presented will be the same as the jitter on the signals instructing frames to be presented (coming out for the controller) and that the controller is free to add up to an extra 5ms of jitter on top of the possible 10ms caused by transmission over the binding. The changes to the specification are localised to the QoS profile for the video window and are altered to that shown below (the last two quality statements are different):

```
QosProfile for VideoWindow {
  exp: quality { throughput(1000, c_video.video) >= 25; };
  obl: quality {
        throughput(1000, i_video.videoPresented) >= 25;
        latency(c_video.video, i_video.videoReady) = 0;
        latency(c_video.video, i_video.videoPresented) <= 10;
        latency(r_present.videoPresent, i_video.videoPresented) = 5;
        anchoredJitter(i_video.videoPresented) < 15;
       };
}
```

6 Conclusion

We have demonstrated that it is possible to form a specification that has sufficient precision to enable formal analysis. We also show how to feedback the results of analysis to designers in a way that enables them to make appropriate changes.

In this section we firstly discuss limitations of our approach to designing distributed systems and the performance analysis technique. In addition we discus related work from the community that addresses similar problems. Finally we conclude with a look at future possibilities and a summary of the work presented.

6.1 Limitations

There are two aspects to the work presented in this paper, the approach taken for designing distributed systems and the technique for verifying the QoS of the system. We evaluate each of these aspects in turn.

Structurally, the design approach is limited only by the confines of the ODP framework. The proposed languages enable the definition of all structural components from the computational viewpoint. This does only give a computational specification; to form a complete system specification it is necessary to provide specifications from the other four ODP viewpoints. However, that would not be within the scope of this paper.

The specification of behaviour is limited by the state diagram formalism we adopt and the requirement to be able to map such specifications to timed automata. Behaviours that cannot be specified as timed automata cannot be specified. Another likely problem is that some behaviour may not be best suited for specification as a state diagram.

The specification of QoS is limited (again) by the nature of the language chosen to express it, CQML. The primary limitation we discovered with this language is within the manner that we use it. In the context of an object we use it to define constraints on interactions at each interface supported by an object; if an object supports an interface signature multiple times – e.g. a multi-cast binding to distribute a stream from source to multiple consumers – there is no means in CQML to quantify over the collection of interfaces. Although being a limitation of the original definition of CQML, there is no apparent reason why it could not be extended to enable such quantifications.

A more specific limitation is imposed by our analysis technique; we currently only address the analysis of specifications on the three QoS characteristics, latency, anchored jitter and throughput; there are potentially many other characteristics that could be constrained. Additionally, the use of timed automata and in particular the UPPAAL variety of timed automata puts some limitations on the nature of the constraints on these characteristics. For instance, we cannot handle statistical constraints such as restrictions on the average throughput, or percentage based constraints such as 90% of the time the average jitter must be < some value. In fact, as described in a previous section, we in actuality don't verify against throughput, we verify against a minimum inter-arrival time.

6.2 Related Work

The design approach presented in this paper is an evolution of our earlier work [14, 15]. Previously we proposed an approach that used the UML in a stricter fashion for the structural and behavioural design and the language QL for specifying QoS. The strict use of UML caused designs to be expressed in a manner that made them hard to read and lengthy to write. Additionally, the QoS language QL does not integrate well with UML – as described in [15].

To improve upon this, we have adopted a more flexible approach to using UML, providing stereotypes to enable design using specifically the concepts defined in the RM-ODP. This approach has also been used in [33] although they do not use the RM-ODP terminology and in [21] and [23] which stereotype UML elements to define a language for creating ODP Enterprise specifications. There is also the rather heavyweight approach defined in the UML profile for EDOC [34]; we found that the lighter weight approach proposed in this paper is more easily used. We have allowed our design

approach to be influenced by non-UML based methods such as the work at Lancaster [8] and in particular the methods proposed by Blair and Stefani in [18].

Our approach to QoS specification uses the Aagedal's CQML language [5] which he has shown to be well integrated with and useable in the context of UML based designs [28]. There are other approaches to the specification of QoS are discussed in [5] and in [35]. The approach taken by [36] and defined in the new CORBA 3 standard [10] is to use extensions of the OMG's Interface Definition Language (IDL) for defining QoS, however we consider this to be a technology specific approach and prefer the use of a language less related to implementation. Similarly, [37] suggests the use of TINA-ODL [7] that is also and extension of the OMG IDL. Finally, The OMG has issued an RFP for a UML profile for modelling QoS [38]; in [28] the authors state that CQML is intended to contribute towards this RFP.

Our verification technique builds on previous works that propose the use of Timed Automata for modelling QoS and Statechart based behaviour specifications. [39] map UML state machines into UPPAAL as does [40]; either of these techniques would compliment our work, giving us a mechanism to transform the state diagram based functional behaviour into UPPAAL timed automata. A group at Lancaster have made significant use of timed automata for modelling distributed and multimedia systems: [41] discusses the use of temporal logic and timed automata for modelling such systems in a multi-paradigm approach and in [42] they specify systems directly in TA and map these to Java-beans to provide prototype implementations of the specified systems. The work of [32] shows ways to model QoS using timed automata and we have made use of that work to compliment our own.

Alternative techniques have been investigated, such as Process Algebra [43], Simulation based techniques [13] and Temporal Logics [44].

6.3 Future Work

Currently, the generation and transformation of specifications is carried out manually; there are plans to automate these processes as part of future work. In particular we intend to automate the transformation of UPPAAL event traces into snapshot series' and the generation of timed automata templates from CQML statements.

Additionally we are looking at techniques for inferring QoS constraints across interface bindings and through an objects functional behaviour. It is expected that this will enable us to analyse individual components of a particular configuration, separately - deducing similar results as if we had analysed the complete configuration. This should aid us in avoiding the huge state explosion problems we would get by attempting to model check large timed automata networks.

6.4 Summary

This paper has demonstrated a technique for generating a network of timed automata from the computational viewpoint design of a multi-media system. The computational viewpoint design includes the specification of both functional and non-functional aspects of behaviour. This is in the form of UML state diagrams for the functional behaviour and CQML statements to define the non-functional QoS of the system. The overall structure of the system and its components is specified using stereotyped UML class diagrams to give template specifications and snapshot diagrams to define particular configurations of objects.

We make use of existing techniques for mapping UML state diagrams on to UPPAAL style timed automata templates. To map the QoS statements, we defined timed automata templates that model the three QoS characteristics – latency, throughput and anchored jitter; these templates are instantiated using parameter values taken from the CQML statements.

A computational snapshot diagram (e.g. Figure 1) is used to deduce a particular network of parallel automata constructed from the generated templates. This network can be model checked (using UPPAAL) to give feedback as to whether or not the functional behaviour verifiably conforms to the specified QoS. The event traces of UPPAAL can be used to construct a series of snapshots that illustrate the sequence of actions that lead to a problem. This enables the feedback to be in a form of the original design language rather than in the "lower level" analysis language.

7 References

[1] ITU-T Recommendation X.901-5 10746-2 to 5:1996-99, Information Technology - Open Distributed Processing - Reference Model: All Parts

[2] ITU-T Draft Recommendation X.930 (1998) | ISO/IEC JTC1/SC7 N2013:1998, Information Technology - Open Distributed Processing - Interface references and Binding

[3] ITU-T Recommendation X.642 (1998) | ISO/IEC 1998, Information technology - Quality of service - Guide to methods and mechanisms
[4] ITU-T Recommendation X.641 (1997) | ISO/IEC 13236:1997, Information technology - Quality of service: Framework
[5] J. Ø. Aagedal, "Quality of Service Support in Development of Distributed Systems," PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, The University of Oslo, 2001
[6] P. Hoschka, "Synchronized Multimedia Integration Language (SMIL) 1.0 Specification," WC3 REC-smil-19980615, June 1998.
[7] TINA, "TINA Object Definition Language (TINA-ODL) Manual," TINA Consortium Document No. TR_NM.002_1.3_95, June 1995.
[8] G. Blair, L. Blair, and J.-B. Stefani, "A Specification Architecture for Multimedia Systems in Open Distributed Processing," *Computer Networks and ISDN Systems, Special Issue on Specification Architecture*, vol. 29, pp. 473-500, 1997.
[9] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken, "Specifying and Measuring Quality of Service in Distributed Object Systems," in proceedings First International Symposium on Object-Oriented Real-time Distributed Computing (ISORC 98), pp. 43-52.
[10] OMG, "Common Object Request Broker Architecture (CORBA/IIOP), version 3," Object Management Group formal/2002-11-03, November 2002.
[11] J. Siegel, *CORBA 3 Funcementals and Programming*, second ed: John Wiley & Sons, Inc., ISBN 0-471-29518-3, 2000.
[12] OMG, "The Unified Modeling Language Version 1.4," Object Management Group formal/01-09-67, 2001.
[13] A. G. Waters, P. F. Linington, D. H. Akehurst, P. Utton, and G. Martin, "Permabase: Predicting the performance of distributed systems at the design stage," *IEE Proceedings - Software*, vol. 148, pp. 113-121, August 2001.
[14] B. Bordbar, J. Derrick, and A. G. Waters, "A UML Approach to the Design of Open Distributed Systems," in C. George and H. Miao (eds) proceedings Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002, Springer, Lecture Notes in Computer Science, 2495, Shanghai, China, pp. 561-572, October 2002.
[15] B. Bordbar, J. Derrick, and A. G. Waters, "Using UML to specify QoS constraints in ODP," *Computer Networks*, vol. 40, pp. 279-304, 2002.
[16] A. Rajeev and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183-235, April 1994.
[17] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Springer International Journal of Software Tools for Technology Transfer*, vol. 1, October 1997.
[18] G. Blair and J.-B. Stefani, *Open Distributed Processing and Multimedia*: Addison Wesley, ISBN 0-201-17794-3, 1997.
[19] P. F. Linington, "RM-ODP: The Architecture," in K. Raymond and E. Armstrong (eds) proceedings Open Distributed Processing: Experience with Distributed Environments, 3rd IFIP TC 6/WG 6.1 International Conference on Open Distributed Processing, Chapman and Hall, February 1995.
[20] J. R. Putman, *Architecting with RM-ODP*: Prentice Hall, ISBN 0-13-019116-7, 2001.
[21] P. F. Linington, "Options for expressing ODP Enterprise Communities and their Policies by using UML," in proceedings 3rd International Conference on Enterprise Distributed Object Computing (EDOC99), IEEE, Silver Spring, pp. 72-82, September 1999.
[22] M. Belaunde and J.-M. Cornily, "Specifying Distributed Object Applications Using the Reference Model for Open Distributed Processing and The Unified Modeling Language," in proceedings 3rd International Conference on Enterprise Distributed Object Computing (EDOC99), IEEE, Silver Spring, September 1999.
[23] M. Steen and J. Derrick, "ODP Enterprise Viewpoint Specification," *Computer Standards and Interfaces*, vol. 22, pp. 165-189, September 2000.
[24] B. Bordbar, J. Derrick, and A. G. Waters, "Using UML to specify QoS constraints in ODP," *Computer Networks and ISDN Systems*, 2001.
[25] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-274, 1987.
[26] D. Harel, "On visual formalisms," *Communications of the ACM*, vol. 31, pp. 514-539, 1988.
[27] OMG, "Response to the UML 2.0 OCL Rfp (ad/2000-09-03)," Object Management Group ad/2002-05-09, 2002.
[28] J. Ø. Aagedal and E. F. Ecklund, "Modelling QoS: Towards a UML Profile," in J.-M. Jezequel, H. Hussmann, and S. Cook (eds) proceedings <<UML>> 2002 The Unified Modeling Language: Model Engineering, Concepts, and Tools, Springer, LNCS, LNCS 2460, Dresden, Germany, pp. 275-289, October 2002.
[29] J. Ø. Aagedal and A. Berre, "ODP-Based QoS-Support in UML," in proceedings First International Enterprise Distributed Object Computing Workshop (EDOC'97), 1997.
[30] ITU-T Recommendation X.642 (1998) | ISO/IEC 13243:1999, Information technology - Quality of service - Guide to methods and mechanisms
[31] A. David and M. O. Moller, "From HUppaal to Uppaal: A translation from hierarchical timed automata to flat timed automata," BRICS, Department of Computer Science, University of Aarhus, Research Series RS-01-11, March 2001.

[32] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink, "Automatic verification of a lip-synchronisation algorithm using uppaal - extended version," in B. Luttick, J. F. Groote, and J. V. Wamel (eds) proceedings FMICS'98 Third International Workshop on Formal Methods for Industrial Critical Systems, CWI, Amsterdam, The Netherlands, pp. 97-124, May 1998.
[33] M. Born, M. Holz, and M. Kath, "A Method for the Design and Development of Distributed Applications using UML," in proceedings International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific), IEEE Computer Society Press, Sydney, Australia, November 2000.
[34] OMG, "UML Profile for Enterprise Disributed Object Computing," Object Management Group ptc/02-02-05, Febuary 2002.
[35] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A survey of QoS architectures," *Multimedia Systems*, vol. 6, pp. 138-151, 1998.
[36] D. G. Waddington, G. Coulson, and D. Hutchison, "Specifying QoS for Multimedia Communications within Distributed Programming Environments," in G. Ventre, J. Domingo-Pascual, and A. Dantine (eds) proceedings Multimedia Telecommunications and Applications, Third International COST 237 Workshop, Springer, Lecture Notes in Computer Science, 1185, Barcelona, Spain, pp. 75-103, November 1996.
[37] P. Leydekkers and V. Gay, "ODP View on Quality of Service for Open Distributed Multimedia Environments," in A. Vogel and J. d. Meer (eds) proceedings 4th International IFIP Workshop on QoS (IWQOS'96), Paris, France, March 1996.
[38] OMG, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, Request for Proposal," Object Management Group ad/02-01-07, January 2002.
[39] A. Knapp, S. Merz, and C. Rauh, "Model Checking Timed UML State MAchines and Collaborations," in W. Damm and E.-R. Olderog (eds) proceedings 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, FTRTFT 2002, Co-sponsored by IFIP WG 2.2,, LNCS, Volume. 2469, Oldenburg, Germany, September 2002.
[40] A. David, M. O. Moller, and W. Yi, "Formal Verification of UML Statecharts with Real-Time Extensions," in R.-D. Kutsche and H. Weber (eds) proceedings 5th International Conference, FASE 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Springer, LNCS, Volume 2306, Grenoble, France, April 2002.
[41] L. Blair, "The Role of Temporal Logic and Time Automata in Distributed Multimedia Systems," in proceedings Modal & Temporal Logic Based Planning for Open Networked Multimedia Systems (PONMS '99), Cape Cod, MA, pp. 1-7, November 1999.
[42] T. Jones and L. Blair, "Prototyping of Real-time Component Based Systems by the use of Timed Automata," in P. Hofmann and A. Schürr (eds) proceedings Workshop on Object-oriented Modeling of Embedded Real-time Systems (OMER-2 '01), GI-Edition - Lecture Notes in Informatics (LNI), P-5, Munich, Germany, May 2001.
[43] H. Bowman, J. Bryans, and J. Derrick, "Analysis of a Multimedia Stream using Stochastic Process Algebra," *The Computer Journal*, vol. 44, 2001.
[44] G. Graw, P. Herrmann, and H. Krumm, "Verification of UML-based real-time system designs by means of cTLA," in proceedings 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC2K), IEEE Computer Society Press, Newport Beach, pp. 86-95, 2000.