Design and Verification of Distributed Multi-media Systems

David Akehurst, John Derrick, and A. Gill Waters

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

Abstract. Performance analysis of computing systems, in particular distributed computing systems, is a complex process. Analysing the complex flows and interactions between a set of distributed 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.

1 Introduction

This paper demonstrates a specific approach to the design of distributed systems. The approach enables verification that Quality of Service (QoS) [15] specifications are met by the specified behavioural aspects of the design. The context which we aim to support is that of designers using the UML/OCL paradigm but requiring stronger verification than these would traditionally facilitate. This leads to an approach with design based around an adaptation of the UML supported by verification in a more formal setting provided by model checking timed automata.

The 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 and a variation on UML object diagrams (snapshots) is used to define particular configurations of objects.

This approach to design enables us to verify that the environmental contract of an object (defined by the QoS specification) is met by the defined functional behaviour of that object. The verification is enabled by generating a network of timed automata (TA) from the design, which makes use of existing techniques for mapping UML state diagrams on to UPPAAL style timed automata templates. To map the QoS statements, we define 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 snapshot diagram (e.g. Fig. 1) is used to deduce a particular network of parallel automata constructed from the generated TA 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.

The approach to design and verification described in this paper, builds on our work from a previous project [14], which address performance prediction from UML system designs and earlier work from this project [6], regarding the UML and specification of QoS. In particular we build on the work of [3], which describes a modelling language for the Computational Viewpoint that is an adaptation of the UML to enable design of distributed systems using the concepts defined in the RM-ODP. The work presented in this paper discusses an approach for mapping these high-level design specifications on to TA in order to verify the QoS aspects of the design. [6] deals with the static aspects of specifying QoS, the current work extends this by addressing the implication of the specified QoS on dynamic aspects of the system.

The rest of this paper is organised as follows. Section 2 illustrates our design approach using an example system. Section 3 describes the translation from specifications in our design languages into Timed Automata for the structure, behaviour and QoS of the system. Section 4 demonstrates the analysis technique incorporating the use of the UPPAAL model checker and illustrates how results can be fed back to the designer. Section 5 discusses some related work and concludes the paper.

2 An ODP Computational Specification

In this section we place our design approach in the context of the ODP framework and illustrate our approach to computational viewpoint design using an adaptation of the UML. Subsequently, in section 3, we will show how to translate these designs to TA in order to verify aspects of the design.

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 current trend in software design communities is to make use of the UML as a specification language for system design. However, for the purpose of designing distributed systems (in particular within the context of the ODP computational viewpoint) the UML is deficient in a number of ways. To combat these deficiencies we have adapted the UML design language; this adaptation is presented and discussed in [3]. The description of the following example design highlights some of these adaptations.

In general, our design approach is to start with a snapshot of the system, to give 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 QoS constraints. As our illustrative example we take the specification of a lip synchronisation system, based on the specification presented in [4].

2.1 System Snapshot

A key aspect of a computational viewpoint specification is the decomposition of the system into distributable *objects* that interact at *interface*s. An object, which may be a composition of two or more other objects, is a unit of distribution and management that encapsulates behaviour [16]. To differentiate from the UML concept of an object, we shall use the term *computational object*.

Fig. 1 depicts a computational viewpoint snapshot of an example mobile videophone, including a lip synchronisation component. There are three aspects of the example system – transmitter device, channel, receiver device. 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 *vidCtrl* interface, which is received by the *vidWindow*. Finally, the s*yncController* has an interface *appControl* that is used to reset the synchronisation algorithm.

A UML object diagram could have been used to illustrate the system configuration, but the notation would not effectively distinguish between computational objects, interfaces and bindings. To rectify this, we define an alternative notation which does distinguish between these concepts, enabling the presentation of computational viewpoint snapshots in a clearer fashion. Using the UML object diagram notation, the components would all appear as boxes, however, we use different syntactic representations for each category of component. This syntactic representation is linked to UML descriptions via a metamodel which effectively defines a transformation between the two representations [3].

In the adapted notation, circles depict computational objects. 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. *vidCtrl*) or placed head to head (all other bound interfaces in this snapshot – *transVideo*, *recVideo*, *transAudio*, *recAudio*, *vidReady*, *audReady* and *appControl*).

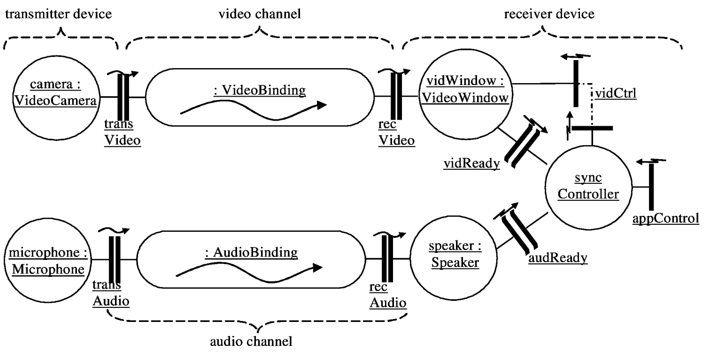


Fig. 1. Computational Viewpoint snapshot illustrating the Lip Synchronisation system

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. As with the UML object diagram notation, where appropriate, we need not give both template and instance name. In this snapshot the two bindings (for audio and video) are labelled with only the template name, the interfaces are labelled with only an instance name, as is *syncController*; other objects are labelled with both template and instance name. Object templates and interface signatures are specified in detail using a notation based on UML class diagrams as described in the next subsection.

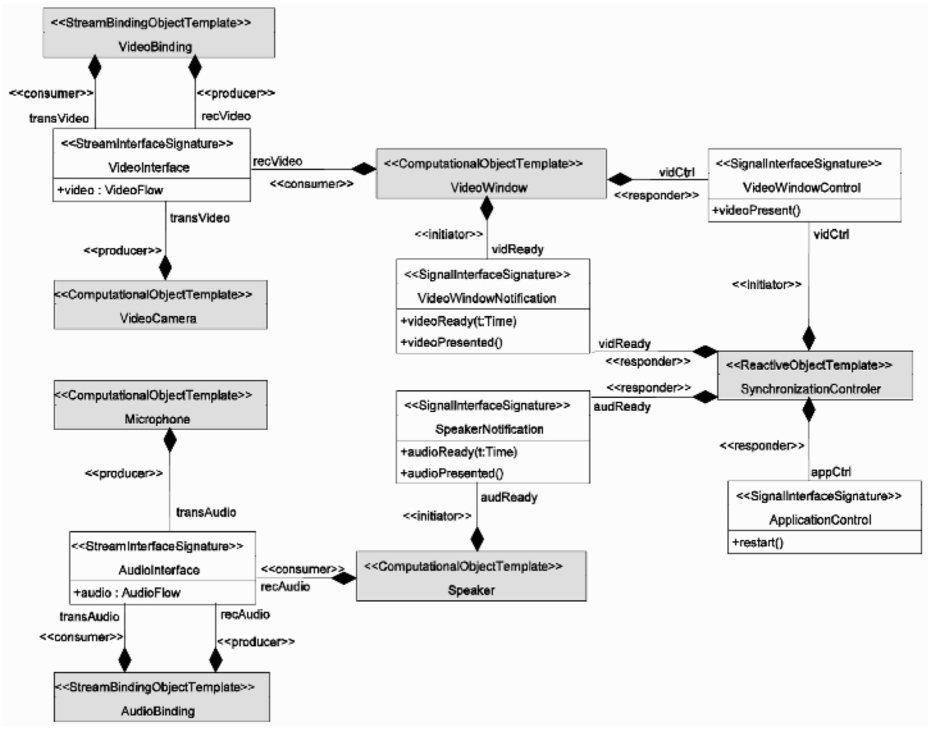
2.2 Template and Signature Specifications

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.

The UML concept of class has a similarity to the ODP notion of template (and signature), thus, we define stereotypes of the UML class concept for specifying components based on 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.

Fig. 2 defines a template diagram for the instances specified in the computational snapshot shown in Fig. 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 specified 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).



<<StreamBindingObjectTemplate>> VideoBinding
<<consumer>> transVideo
<<producer>> recVideo
<<StreamInterfaceSignature>> VideoInterface +video : VideoFlow
recVideo <<consumer>>
<<ComputationalObjectTemplate>> VideoWindow
vidCtrl <<responder>>
<<SignalInterfaceSignature>> VideoWindowControl +videoPresent()
transVideo <<producer>>
<<ComputationalObjectTemplate>> VideoCamera
<<initiator>> vidReady
<<SignalInterfaceSignature>> VideoWindowNotification +videoReady(t:Time) +videoPresented()
vidCtrl <<initiator>>
vidReady <<responder>>
<<ReactiveObjectTemplate>> SynchronizationControler
<<responder>> audReady
<<ComputationalObjectTemplate>> Microphone
<<SignalInterfaceSignature>> SpeakerNotification +audioReady(t:Time) +audioPresented()
<<responder>> appCtrl
<<SignalInterfaceSignature>> ApplicationControl +restart()
<<producer>> transAudio
audReady <<initiator>>
<<StreamInterfaceSignature>> AudioInterface +audio : AudioFlow
<<consumer>> recAudio
<<ComputationalObjectTemplate>> Speaker
transAudio <<consumer>>
recAudio <<producer>>
<<StreamBindingObjectTemplate>> AudioBinding

Fig. 2. Template Diagram for the Lip Synchronisation system

Given this specification, the objects and interfaces from the snapshot diagram are related to the appropriate template or signature. In our example, the snapshot already shows the relationship for all of the objects except the *syncController*, which is instantiated from the *SynchronisationController* template; the interfaces are instantiated from the defined signatures with a similar name. The snapshot diagram could be refined to show these relationships but we do not illustrate the refinement here.

2.3 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 how UML State Diagrams can be adapted to specifying behaviour within the computational viewpoint and subsequently illustrates the technique by defining the behaviour of computational objects instantiated from the *SynchronisationController* template (i.e. *syncController*). It is assumed that the reader is familiar with standard State Diagrams (or *Statecharts* [10]).

Normally a State Diagram is associated with a UML class; it is used to specify the behaviour of objects instantiated from that class. Events that affect the state of the object and actions caused by the object are related to interactions (i.e. operations) defined on the class and other classes it is associated with.

In our model for design, interactions are defined on interfaces attached to an object, thus we cannot directly reference those interactions from the object. In addition, a reference to the interaction is not sufficient (on its own) to identify the cause of the event; an interface signature (and hence the interactions defined in it) may be instantiated multiple times for a single object.

In order to use State Diagrams in the context of our adapted design language, we associate a State Diagram with a particular computational object template; however, we have to alter the interpretation of the text specified on transitions.

Normally a transition is labelled with a string that has the following general format:

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

Where, for standard State Diagrams, *event-signature* describes an event with its arguments; the *guard* condition is a Boolean expression written in terms of parameters of the triggering event and attributes of the object. The *action-expressions* are executed if and when the transition fires. Typically an action expression alters the local state of an object or causes another event to be fired, possibly by sending a message to another object.

Our adaptation requires the *event-signature* to identify the interface from which the interaction causing the event is received, along with the identifier for the interaction; *guard* conditions are interpreted in the standard manner; and *action-expressions* either:

1. Alter the local state of the object;
2. Instantiate interfaces to be offered by the object; or
3. Cause interactions at a specified interface; by identification of an interface and an interaction available at that interface.

This alteration to the text label of a transition enables State Diagrams to be used in the context of our adaptation to the UML design language.

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 [4], it is not the purpose of this paper to describe a synchronisation algorithm, we simply use it as an example. A State Diagram (Fig. 3) is presented that specifies the behaviour.

Note that events are identified by an interface name and name from the signature of that interface, e.g. `appCtrl.restart`. Additionally, signals are sent within an action expression by identifying an interface name and a name from the signature of that interface, e.g. `vidCtrl.videoPresent`.

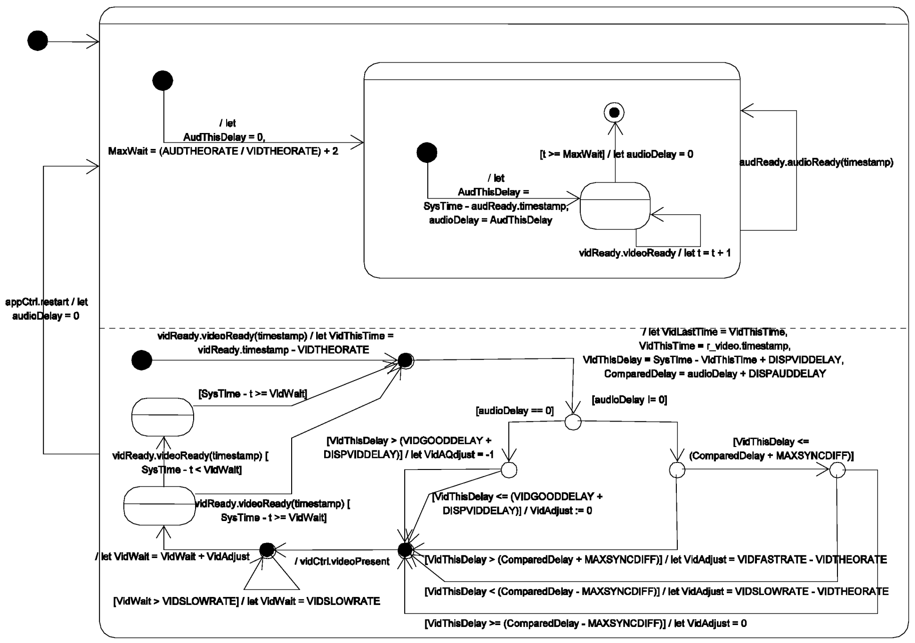

/ let
AudThisDelay = 0,
MaxWait = (AUDTHEORATE / VIDTHEORATE) + 2
[t >= MaxWait] / let audioDelay = 0
audReady.audioReady(timestamp)
/ let
AudThisDelay =
SysTime - audReady.timestamp,
audioDelay = AudThisDelay
vidReady.videoReady / let t = t + 1
appCtrl.restart / let
audioDelay = 0
vidReady.videoReady(timestamp) / let VidThisTime =
vidReady.timestamp - VIDTHEORATE
/ let VidLastTime = VidThisTime,
VidThisTime = r_video.timestamp,
VidThisDelay = SysTime - VidThisTime + DISPVIDDELAY,
ComparedDelay = audioDelay + DISPAUDDELAY
[SysTime - t >= VidWait]
[audioDelay == 0]
[audioDelay != 0]
[VidThisDelay > (VIDGOODDELAY +
DISPVIDDELAY)] / let VidAQdjust = -1
[VidThisDelay <=
(ComparedDelay + MAXSYNCDIFF)]
vidReady.videoReady(timestamp) [
SysTime - t < VidWait]
[VidThisDelay <= (VIDGOODDELAY +
DISPVIDDELAY)] / VidAdjust := 0
vidReady.videoReady(timestamp) [
SysTime - t >= VidWait]
/ let VidWait = VidWait + VidAdjust
/ vidCtrl.videoPresent
[VidThisDelay > (ComparedDelay + MAXSYNCDIFF)] / let VidAdjust = VIDFASTRATE - VIDTHEORATE
[VidThisDelay < (ComparedDelay - MAXSYNCDIFF)] / let VidAdjust = VIDSLOWRATE - VIDTHEORATE
[VidWait > VIDSLOWRATE] / let VidWait = VIDSLOWRATE
[VidThisDelay >= (ComparedDelay - MAXSYNCDIFF)] / let VidAdjust = 0

Fig. 3. State Diagram specification of the Synchronisation Controller's behaviour

2.4 Environment Contracts – QoS Specification

The previous subsections have defined the structure, templates and functional behaviour of the system. We now 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, which it uses to describe QoS constraints. This is a contract between an object and its environment, i.e. all other objects with which it interacts. As interactions occur across interfaces, environment contacts for an object generally involve one or more interfaces. A QoS 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)". The expectations and obligations are expressions that constrain the QoS characteristics of interactions with the object.

There are many possible QoS characteristics that could be constrained. 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 (e.g. variation in rate of flow of frames).

There is currently no clear contender for a most commonly used (de facto) QoS specification language. One that we have found to be most suited to our design approach, partly due to its association with the Object Constraint Language (OCL) and UML, is the Component Quality Modelling Language (CQML) [1]. CQML is a lexical language for QoS specification and has been developed to explicitly include as many features as possible. We have found the language to be expressive, very useable and easily adapted to integrate with our other UML based languages used within the ODP framework. The CQML semantics require that the interaction constrained by a characteristic must facilitate access to a historical sequence of events. The definition of *quality characteristic*s, such as latency, throughput and anchored jitter, is expressed in terms of the history of events (we do not provide the definition of these characteristics in this paper). Constraints regarding particular characteristics are formed in CQML by specifying *quality statement*s; these are grouped to form QoS specifications on particular components 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 a particular interaction. 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 Fig. 2 can be defined. We illustrate this (below) using the *VideoBinding* component. QoS specifications for other parts of the system can be found in [2].

Use of CQML, in the context of our model for design, means that QoS profiles are associated with computational objects (or their templates) and the constrained interactions are identified by reference to an interface and an appropriate interaction. The requirement to provide event sequences is met by the 'Event Notification Function', defined in the RM-ODP [16], which also requires event histories to be made available. We alter the standard CQML notation slightly, changing the keywords *profile*, *uses* and *provides* into *QoSProfile*, *exp:* and *obl:* as we find this allows the expressions to be more easily understood in the context of the ODP framework terminology; we also do not require profiles to be named.

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,transVideo.video)>=25; };
  obl: quality {
   throughput(1000,recVideo.video)>=25;
   latency(transVideo.video,recVideo.video).maximum=60;
   latency(transVideo.video,recVideo.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 *transVideo*. 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 (*recVideo*) supported by the binding in the role of a *producer*. 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.

The particular *VideoFlow* interactions, on which the constraints are placed, are navigated to using the association end names of the associations relating object templates to interface signatures.

3 Translation to Timed Automata

To verify that the constraints defined by the QoS specifications are met by the functional behaviour, for a particular configuration of system components, we create a network of parallel timed automata. It is intended that the TA network be automatically generated by transforming the information contained in the design specification. There are three aspects to this transformation:

1. State Diagrams are mapped to hierarchical timed automata [8] which can be flattened for input to UPPAAL using the method described in [8].
2. The parameters of the QoS constraints are used as arguments to TA templates that model the appropriate QoS characteristic.
3. A snapshot configuration diagram is used to appropriately connect the set of TAs, produced by the first two steps, into a single network.

This gives a network of TA that model the whole system, plus the environment in which the system is supposed to execute. There are three different sets of automata arising from the transformation:

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 [13], which is subsequently used to check for deadlock in the system. Formally, there are six situations which could cause deadlock: internal deadlock in an automaton 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.

In short, a deadlock is due either to an internal problem of the functional behaviour of a computational object; or due to the computational object not interacting with its environment in the manner specified by 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 section 4 by analysis of our example system. Before we discuss the analysis, the following subsections describe the manner in which we have modelled the three QoS characteristics – latency, throughput and jitter – using Timed Automata. These automata are instantiated by giving values to their parameters, taken from the CQML specifications.

3.1 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 the two characteristics and we can define a single TA that models both. Anchored jitter is defined as a variation in 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 times 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.

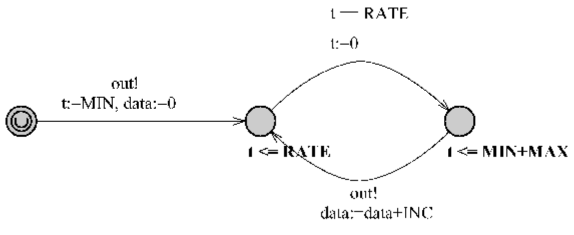


Fig. 4. Expectation Anchored Jitter and Throughput

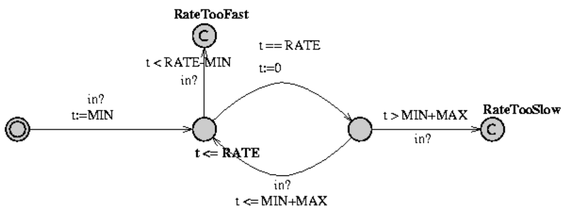


Fig. 5. Obligation Anchored Jitter and Throughput

A QoS expectation relating to throughput and anchored jitter can be encoded as the TA illustrated in Fig. 4. 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.

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 of the example, which specifies a constraint on the total number of frames arriving within a specified time (i.e. 1000 milliseconds). Strictly speaking this TA is modelling the quality statement:

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

Similarly, the TA shown in Fig. 5 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.

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.

3.2 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, the UPPAAL 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. Therefore, 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 calculate this value, called 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 Fig. 6 models 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.

4 Performance Analysis

This section first describes the result of verifying our illustrative example, followed by a description of a technique to present back to a designer, in the context of the original design language, a trace originally produced by UPPAAL in terms of the generated automata.

4.1 Verification of the Example

As discussed above, the UPPAAL verifier is used to check the model against a query stating that there are no deadlocks for all time: `A[] not deadlock`

For our example, the UPPAAL verifier indicates that there is a deadlock; its cause is that the QoS specified for the video window is not met by its behaviour, i.e. situation (b) of those described in section 3. UPPAAL provides a trace of Automata events that lead to the deadlock situation; these events are succinctly described as follows:

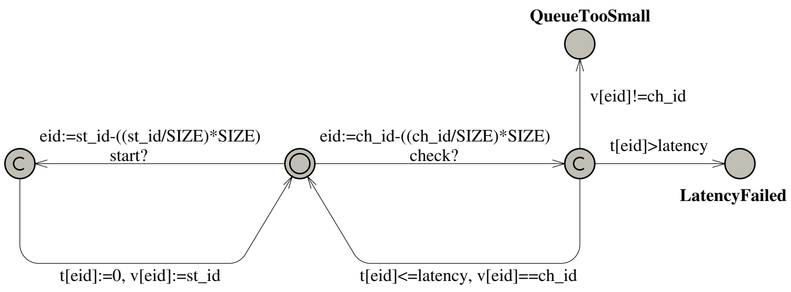


Fig. 6. An Automaton Template for Latency Obligation

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 anchoredJitter QoS constraint on the video window.
7. Analysis of this trace determines that the QoS specified for the video window constrains it 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.

After analysing the 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.

For example, we could implement options 2 and 3. To do so we could a fix 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, which is altered to that shown below. Using this, the resulting system is now free from this deadlock; absence of all such deadlocks indicates that the functional behaviour is consistent with the QoS specification.

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

4.2 Feedback of Results

A weakness of model checking TA in the context of higher level languages, such as the UML, 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 occur or were taken. A trace in this form is of little use to a designer working in the UML/OCL context who is not familiar with the level of formality provided by TA; 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 (derived from UPPAAL) 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.

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 primitive bindings between interfaces map to the synchronisation channels in UPPAAL; thus the trace of events given by UPPAAL 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 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 Fig. 7. This transformation appears to be automatable, and ongoing work is investigating implementation of an automatic transformer.

Table 1. UPPAAL Trace of transitions leading to deadlock

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**

5 Conclusion

Our verification technique builds on previous works that propose the use of Timed Automata for modelling QoS and Statechart based behaviour specifications. [12]

maps UML state machines into UPPAAL as does [9]; either of these techniques would compliment our work, and give us mechanisms 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: [5] discusses the use of temporal logic and timed automata for modelling such systems in a multi-paradigm approach and in [11] they specify systems directly in TA and map these to Java-beans to provide prototype implementations of the specified systems. The work of [7] shows ways to model QoS using timed automata and we have made use of that work to compliment our own.

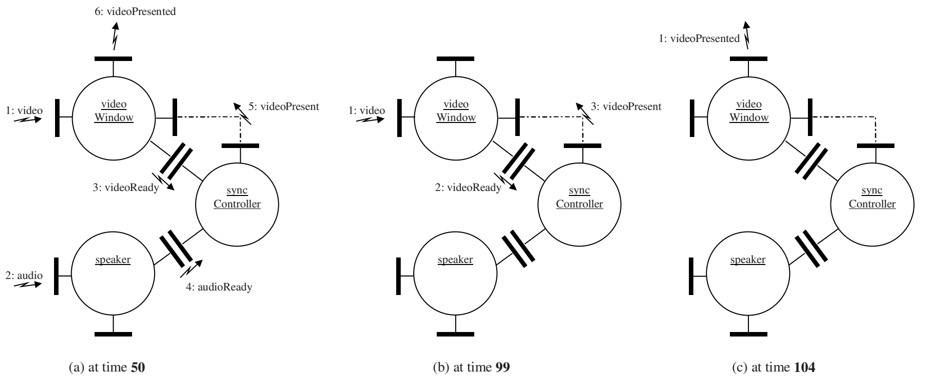


Fig. 7. Series of snapshots leading to deadlock / invalid QoS

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.

Structurally, the proposed design languages enable the definition of all structural components from the computational viewpoint; to form a complete system specification it is necessary to provide specifications from the other four ODP viewpoints; this is left as future work.

Of course there are some limitations to the approach. For example, the specification of QoS is limited by the nature of the language chosen to express it, CQML. The primary limitation we discovered with this language, is that in the context of a computational 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, there is no means in CQML to quantify over the collection of interfaces; we believe that CQML could be extended to enable such quantifications.

Future plans include provision of a design tool that incorporates our design languages and facilities to perform the verification and feed back of results. 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.

References

1. Aagedal J. Ø. "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
2. Akehurst D. H., Bordbar B., Derrick J., and Waters A. G., "Design and Verification of Distributed Multi-media Systems," University of Kent at Canterbury, 1-03, January 2003.
3. Akehurst D. H., Derrick J., and Waters A. G., "Addressing Computational Viewpoint Design," in proceedings EDOC 2003, Brisbane, Australia, September 2003.
4. Blair G. and Stefani J.-B., *Open Distributed Processing and Multimedia*: Addison Wesley, ISBN 0-201-17794-3, 1997.
5. Blair L., "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.
6. Bordbar B., Derrick J., and Waters A. G., "Using UML to specify QoS constraints in ODP," *Computer Networks*, vol. 40, pp. 279-304, 2002.
7. Bowman H., Faconti G., Katoen J.-P., Latella D., and Massink M., "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.
8. David A. and Moller M. O., "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.
9. David A., Moller M. O., and Yi W., "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.
10. Harel D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-274, 1987.
11. Jones T. and Blair L., "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.
12. Knapp A., Merz S., and Rauh C., "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.
13. Larsen K. G., Pettersson P., and Yi W., "UPPAAL in a Nutshell," *Springer International Journal of Software Tools for Technology Transfer*, vol. 1, October 1997.
14. Waters A. G., Linington P. F., Akehurst D. H., Utton P., and Martin G., "Permabase: Predicting the performance of distributed systems at the design stage," *IEE Proceedings - Software*, vol. 148, pp. 113-121, August 2001.
15. X.641, "Information technology - Quality of service: Framework," vol. 1997: ITU-T Recommendation, 1998.
16. X.901-5, "Information Technology - Open Distributed Processing - Reference Model: All Parts," ITU-T Recommendation, 1996-99.