Automatic Verification of a Lip Synchronisation Algorithm Using UPPAAL - Extended Version - *

H. Bowman [†] G. Faconti [‡] J-P. Katoen [§] D. Latella [¶]

M. Massink [‖]

Abstract

We present the formal specification and verification of a lip synchronisation algorithm using the real-time model checker UPPAAL. A number of specifications of this algorithm can be found in the literature, but this is the first automatic verification. We take a published specification of the algorithm, code it up in the UPPAAL timed automata notation and then verify whether the algorithm satisfies the key properties of jitter and skew. The verification reveals some flaws in the algorithm. In particular, it shows that for certain sound and video streams the algorithm can timelock before reaching a prescribed error state.

1 Introduction

It is now well recognised that the next generation of distributed systems will be distributed *multimedia* systems, supporting multimedia applications such as, video conferencing. Importantly though, multimedia imposes a number of new requirements on distributed computing, not least of which is the need to ensure "timely" transmission

*This paper has been presented at the 3rd Int. ERCIM/FMICS Workshop, Amsterdam, May 1998, and it can be found in: Jan Friso Groote, Bas Luttik and Jos van Wamel (Eds.) *Proceedings of the Third International Workshop on Formal Methods for Industrial Critical Systems*, CWI, The Netherlands, 1998.

[†]Computing Lab., U. of Kent, Canterbury, Kent, CT2 7NF, UK. Currently on leave at CNUCE under the support of the ERCIM Fellowship Programme.

[‡]CNR-Istituto CNUCE, Via S. Maria 36, 56126, Pisa, Italy.

[§]Lehrstuhl für Informatik VII, U. Erlangen, D-91058 Erlangen, Germany.

[¶]CNR-Istituto CNUCE, Via S. Maria 36, 56126, Pisa, Italy.

[‖]Dept. of Computer Science, U. of York, UK. Supported by the TACIT network under the European Union TMR Programme, contract ERB FMRX CT97 0133.

and presentation of multimedia data, e.g. ensuring that the end-to-end timing delay between transmitting and presenting video frames stays within acceptable bounds. Consequently, there is significant interest in how to determine that multimedia systems satisfy their real-time requirements (which, in the distributed multimedia systems field, are typically categorized as *Quality Of Service (QOS)* properties).

Furthermore, it would be advantageous if the real-time properties of systems could be analysed during the early stages of system development. This prevents the costly scenario of constructing a finished system only to find that it doesn't meet its real-time requirements. Formal specification and verification offers great potential in this respect.

Consequently, a number of researchers have considered techniques for the specification [3, 10, 17, 9] and verification [5] of multimedia systems. One contribution of this body of work is to identify a number of canonical examples of multimedia systems, e.g. a multimedia stream and a lip synchronisation algorithm [3]. The latter is particularly important as it offers a non-trivial example of real-time synchronisation between continuous media.

The lip-sync example was first described in the synchronous language Esterel [18]. Then specifications in a number of different formalisms were presented, e.g. in a timed LOTOS [17], in a dual language approach, LOTOS/QTL, [4, 3] and in Timed CSP [9]. Typically, these specifications describe the algorithm in their chosen formalism and then postulate that it satisfies certain timing requirements. However apart from [9], where some properties are proved by hand, no formal verification of the postulated properties exist.

This paper responds to this deficiency by considering formal verification of the lip-sync algorithm using the *real-time model checker UPPAAL* [12]. The model checking problem is to determine whether a system (usually described as a network of communicating automata) satisfies a particular temporal logic property. In our case, the system will be described in a *timed* automata notation and the properties will be defined in a *timed* temporal logic. Such automatic verification is potentially far more efficient than the complex hand proofs considered in [9].

One of the goals of this paper is to find out for which streams this protocol behaves correctly, i.e. maintains lip synchronisation or signals a proper error when the synchronisation requirements are not met. The performed verification reveals some flaws in the algorithm. In particular, it shows that for certain sound and video streams the algorithm can timelock before reaching a prescribed error state.

Structure of the paper. Section 2 introduces the lip-sync problem. Section 3 introduces the UPPAAL tool suite. Section 4 considers how streams with varying jitter behaviour can be defined. Section 5 discusses the important issue of how to express timeout behaviour in UPPAAL. Section 6 presents the UPPAAL specification of the algorithm. Section 7 considers the results of our verification and section 8 gives a concluding discussion.

2 The Lip Synchronisation Problem

2.1 Background

It is typically argued that the incorporation of multimedia enforces three new requirements on distributed systems:-

- *Continuous Interaction.* Traditionally, distributed systems communication paradigms involve interaction of a logically singular character, e.g. a remote procedure call. However, the advent of multimedia means that this is not sufficient. In particular, interaction of an "ongoing" nature must be provided, e.g. continuous transmission of video frames in a video conferencing application. Such an ongoing interaction is called a *stream* (the term *flow* is also often used [13]).

- *Quality of Service.* QoS requirements have to be associated with such continuous interactions. For example, in a video conferencing application, if the end-to-end latency delay between the generation of frames and their presentation becomes too great the sense of simultaneous interaction will be lost. Typical quality of service properties include: *end-to-end latency* between the generation of packets and their presentation (which we simply call *latency*), *throughput*, i.e. the rate at which packets are presented and *jitter*, which concerns the variability of delay, we consider it further in subsection 2.2.

- *Real-time Synchronisation.* It is also often necessary to synchronise multiple streams; lip-synchronisation is just such an example. Application specific real-time synchronisation also arises, e.g. if captions need to be presented at particular points in a video presentation.

A verification using UPPAAL of a multimedia stream with associated quality of service parameters was presented in [5]. It embraces the first two of the above requirements. Here we build upon this previous work by considering UPPAAL verification in the context of the third requirement.

2.2 Jitter, Drift and Skew

A number of key real-time properties can be used to quantify the quality of synchronisation between audio and video. This subsection introduces these properties. One reason for doing this is to clarify terminology which has previously been used inconsistently.

Jitter. In this paper we will only be concerned with *bounded* jitter, i.e. placing upper and lower bounds on the acceptable level of jitter. In a statistical setting we can also obtain a measure of variability of presentation times by considering the *statistical variance* of latency. However, such an interpretation is beyond the scope of the tools

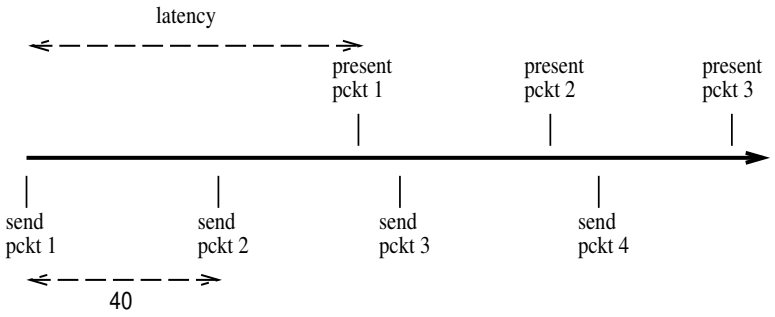


Figure 1: An Optimum Playout of Packets

we have available. In the context of this paper we will refer to bounded jitter as simply jitter.

Two interpretations of jitter can be found in the literature:-

- *Anchored Jitter*[1]. Jitter attempts to quantify the acceptable variation around the optimum presentation time. So, assuming a source which transmits at regular intervals, say every 40ms, ideally (if the latency is constant) the sink should play frames with identical spacing, generating a time line such as that shown in figure 1. Anchored jitter measures the maximum variation relative to these optimum presentation times. We refer to it as anchored because it is anchored to the sequence of optimum presentations. For example, it may allow packets to be presented 5ms before or after the optimum presentation time.

- *Non-anchored Jitter.* In contrast, non-anchored jitter is not defined relative to the time line of optimum presentation, rather variability is measured relative to the presentation time of the previous frame. For example, the property might state:

  > *All frames, apart from the first, must be presented within an interval, say* `[35,45]`*, of the previous frame.*

  Importantly, this interpretation allows the presentation sequence to *drift* out relative to the time line of optimum presentation. For example, if each frame is presented 44ms after the previous frame, we will not invalidate the above property, but each presented frame will incure a drift relative to the optimum; +4 for the second frame, +8 for the third, +12 for the fourth and so on.

[1]The term anchored and non-anchored is ours and to our knowledge cannot be found elsewhere in the literature

The anchored jitter interpretation appears frequently in the multimedia literature, however, much of the previous work on lip-sync has interpreted jitter in a non-anchored fashion [3, 19].

Skew. In line with the terminology in [19] we use the term skew to refer to the time difference between related audio and video items. Thus, while jitter is an intra-stream measure, skew is an inter-stream measure. It categorizes the degree to which the two streams are out of synchronization. So, for example we might have a situation where video is skewed by -80ms relative to the audio, i.e. it lags the audio by 80ms.

2.3 Lip Synchronisation

A common approach to obtaining lip-synchronisation is to multiplex the audio and video streams at the source and demultiplex at the sink, i.e. elements of the two streams are physically combined and a single "composite" stream is transmitted. Such an approach automatically ensures synchronisation of audio and video. However, as pointed out in [19], this approach is not always possible or even wanted since different media types need to be handled by different adapters in the system, e.g. compression hardware. Thus, alternative approaches need to be considered in which audio and video are transmitted as separate streams and synchronisation between audio and video is regenerated at the sink. The algorithm we consider here is such a scenario.

Importantly though, resynchronisation at the sink does not always have to be exact, since it is well known that certain "out of synchronisation" levels can not be perceived by the user. In [19] experiments have been performed to determine bounds on acceptable out of synchronisation levels. Thus, in order to avoid ruling out acceptable implementations (i.e. not to overspecify), the lip-sync algorithm accommodates certain out of synchronisation levels.

The basic system configuration that we consider is shown in figure 2. There are two data sources, a sound source and a video source, which generate a pair of data streams. These streams are received at a *presentation device* (in fact, in our specification we will model the arrival of frames at the presentation device and will abstract away from the behaviour of particular sources). The problem is to ensure that play out of the two streams at the *presentation device* is acceptably synchronised.

The algorithm is implemented using a number of components: *sound* and *video* managers and a *controller*. When a sound packet arrives at the presentation device an *savail* signal is passed to the *Sound Manager*. When appropriate, the *Sound Manager* returns an *spresent* to the *Presentation Device* indicating that the packet can be presented. The *Video Manager* has a corresponding behaviour. The *Controller* contains the body of the lip-sync algorithm. It receives *sready*s (respectively *vready*s) from the *Sound* (respectively *Video*) *Manager*, indicating that a *Sound* (respectively *Video*) packet is ready to be played. The *Controller* then evaluates if and when it is appropriate to play the particular packet. It either returns an *sok* (respectively *vok*)

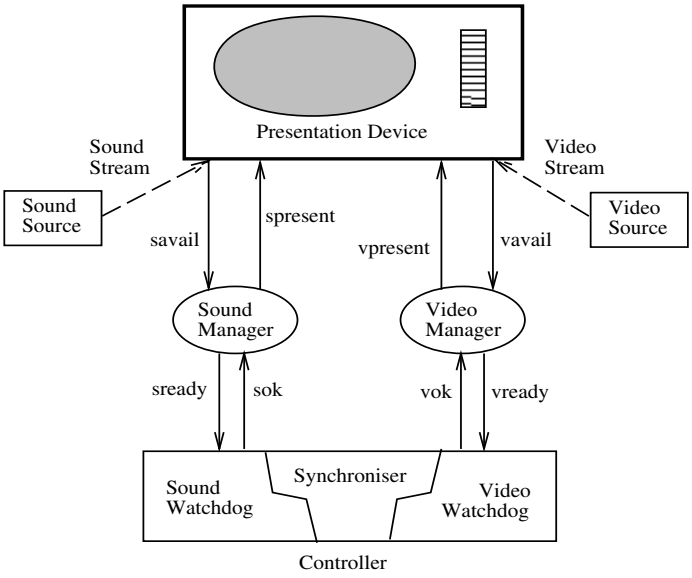


Figure 2: Basic structure of the lip-sync system

at the appropriate time or, if acceptable synchronisation is not recoverable, it signals an error and passes into an error state.

The following requirements characterise acceptable synchronisation between the two streams. Our figures are in line with those used in formal specifications found in the literature [18] [17][2].

- The granularity of time is a millisecond[3].
- A sound packet must be presented every 30ms (each sound packet contains 400 samples of 12khz sampled digital sound). No jitter is allowed on the sound.
- In the optimum, a video packet should be presented every 40ms (i.e. 25 frames per second). However, we allow some flexibility around this optimum:
  - **Non-anchored Jitter** - A video frame must follow the previous video frame by no less than 35ms and no more than 45ms.
  - **Skew** - Video frames may lag sound by no more than 150ms and may precede sound by no more than 15ms .

[2] According to [19] different figures should be used in practice. Although these figures are important they do not affect the essence of the lip synchronisation algorithm.

[3] The algorithm assumes a discrete time solution. Although, UPPAAL supports dense time, in order to stay in line with the existing solutions, we model a discrete time clock in UPPAAL.

One characteristic of the scenario is that there is not a one-to-one correspondence between packets in the two streams. Even at the optimum, sound packets are presented every 30ms and video packets are presented every 40ms. Thus, although we may informally talk about corresponding items in the audio and video stream, this correspondence is not at the level of packets.

3 Introduction to UPPAAL

UPPAAL is a tool-suite for the specification and automatic verification of real-time systems. It has been developed at BRICS in Denmark and at Uppsala University in Sweden. In UPPAAL a real-time system is modelled as a network of (extended) timed automata with global real-valued clocks and integer variables. The behaviour of a network of automata can be analysed by means of the simulator and reachability properties can be checked by means of the model checker. In Figure 3 an overview is given of the different components of the UPPAAL tool and their relation.

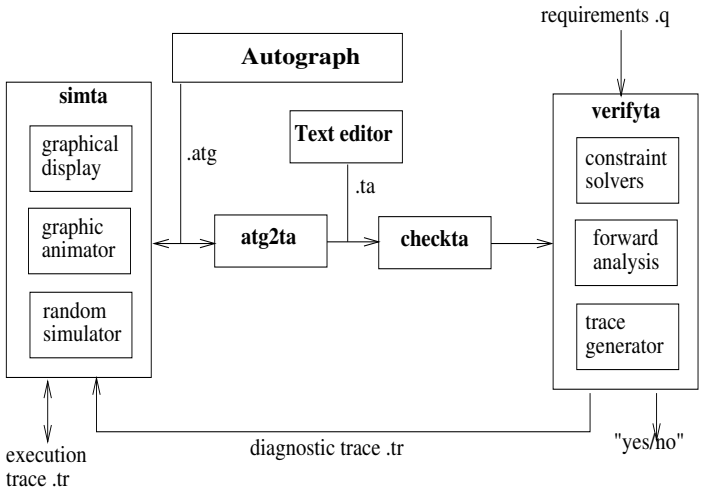


Figure 3: Overview of UPPAAL tool suite

In UPPAAL, automata can be specified in two ways. Graphically by using the tool Autograph or textually by means of a normal text editor. The graphical specification can be used by the graphical simulator 'simta' or be automatically translated into textual form and used as input for the model checker 'verifyta' together with a file with requirements to be checked on the model. The requirements are formulas in a simple temporal logic language that allows for the formulation of reachability properties. The model checker indicates whether a property is satisfied or not. If the property is not satisfied a trace is provided that shows a possible violation of the property. This trace can be fed back to the simulator so that it can be analysed with the help of the graphical

presentation.

3.1 The UPPAAL model

UPPAAL automata consist of nodes and edges between the nodes. Both the nodes, which are called locations, and the edges, which are called transitions, are labelled. A network of automata consists of a number of automata and a definition of the configuration of the network. In the configuration the global real-time clocks, the integer variables, the communication channels and the composition of the network are defined.

The labels on edges are composed of three optional components: a *guard*, an *action* and a number of *clock resets* and *assignments to integer variables*. The *guard* on clocks and data variables expresses under which condition the transition can be performed. Absence of a guard is interpreted as the condition *true*. The synchronisation or internal *action* is performed when the transition is taken. In case the action is a synchronisation action then synchronisation with a complementary action in another automaton is enforced following similar synchronisation rules as in CCS [14]. Absence of a synchronisation action is interpreted as an internal action similar to τ-actions in CCS. The label of a location consists also of three parts: the *name* of the location, an optional *invariant* and optionally the marking **c:**. The invariant expresses constraints on clock values, indicating the period during which control can remain in that particular location. Absence of an invariant is interpreted as the condition *true*. The marking **c:** in front of the location name indicates that the location is *committed*. This option is useful to model atomicity of transition-sequences. When control is in a committed location the next transition must be performed (if any) without any delay and any interleaving of other actions.

In the configuration, the names of the automata which compose the system as well as the global variables and channels are declared. Channels can be declared *urgent*. When a channel is urgent no timing constraints can be defined on the transition labelled by that channel and no invariant can be defined on the location from which that transition leaves. Urgent actions have to happen as soon as possible, i.e. without delay, but interleaving of other actions is allowed if this does not cause delays.

Formally, the states of an UPPAAL model are of the form $(\bar{l}, v)$, where $\bar{l}$ is a *control vector* and v a *value assignment*. The control vector indicates the current control location for each component of the network. The value assignment gives the current value for each clock and integer variable. The *initial state* consists of the initial location of all components and an assignment giving the value 0 for all clocks and integer variables. All clocks proceed at the same speed. There are three types of transitions in an UPPAAL model. An *Internal transition* can occur when an automaton in the network is at a location in which it can perform an internal action. The guard of that transition has to be satisfied and there must be no other transitions enabled that start from a committed location. A *Synchronisation transition* can occur when there are two

automata which are in locations that can perform complementary actions. The guards of both transitions must be satisfied and there must be no other transitions enabled that start from a committed location. A *Delay transition* can occur when no urgent transitions are enabled, none of the current control locations is a committed location and the delay is allowed by the invariants of the current control locations.

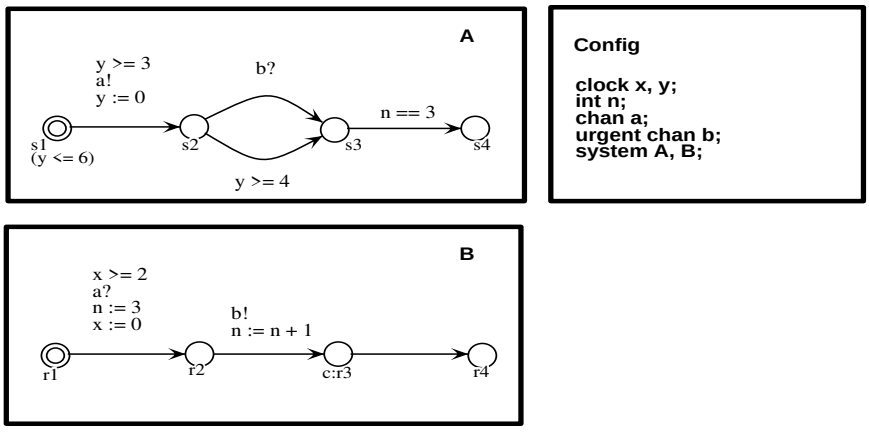


Figure 4: Example of an UPPAAL specification

An example of an UPPAAL specification is given in Figure 4. The transition between *s1* and *s2* can only be taken when the value of clock y is greater than or equal to 3. This holds also for the transition between *r1* and *r2* because the automata A and B are synchronised on channel a. The transition *must* happen before y is equal to 6 because of the invariant at location *s1*. If this invariant would not be there control could have remained in *s1* and in *r1* indefinitely. When control is in *s2* and *r2* the only transition that is possible is the synchronisation on action b. This is because b has been declared as an *urgent* channel in the configuration. Note that if the guard $y >= 4$ would not have been labelling the transition between *s2* and *s3* in A both transitions between those two locations would have been enabled! This is because urgency only prevents the passing of time, but does not prevent the occurrence of other actions that are enabled at the same time. To prevent interleaving actions in this case the location *r2* can be annotated as a committed location. This forces the action b to happen without delay or interference of other actions.

3.2 Simulation and Model Checking

The future behaviour of a network of timed automata is fully characterized by its state, i.e. the control vector $\bar{l}$, and the value of all its clocks and data variables. Clearly this leads to a model with infinitely many states. The interesting observation made by Alur and Dill was that states with the same $\bar{l}$ but with slightly different clock values have runs starting from $\bar{l}$ that are "very similar". Alur and Dill described exactly how to

derive the sets of clock values for which the model shows "similar" behaviour [1]. The sets of clock values are called *time regions*. Regions can be derived from the guards, the invariants and the reset-sets in the UPPAAL model. Since clock variables in the constraints are always compared with integers and because in every model there is a maximum integer with which a clock is compared the state space of a model can be partitioned into finitely many regions. This makes model checking for dense time decidable. In UPPAAL the regions are characterised by simple constraint systems which are conjunctions of atomic clock and data constraints [20].

The properties that can be analysed by the model checker are reachability properties. They are formulas of the following form:

$$\Phi ::= A[]\,\beta \mid E<>\,\beta$$

$$\beta ::= a \mid \beta_1 \text{ and } \beta_2 \mid \beta_1 \text{ or } \beta_2 \mid \beta_1 \text{ implies } \beta_2 \mid \text{ not } \beta$$

where a is an atomic formula of the form: $A_i.l$ where A_i is an automaton and l a location of A_i or $v_i \sim n$ where v_i is a clock or data variable, n a natural number and $\sim$ a relation in $\{<, <=, >, >=, ==\}$. The basic temporal logic operators are, $A[]$ and $E<>$, where, informally, $A[]\,\beta$ requires all reachable states to satisfy β and $E<>\,\beta$ requires at least one reachable state to satisfy β.

Although the final aim of the developers of UPPAAL is to develop a modelling language that is as close as possible to a high-level real-time programming language with various data types the current version is rather restrictive. For example it does not allow assignment of variables to other variables and there is no value-passing in the communication. Despite these restrictions, quite a number of case-studies have been performed in UPPAAL ranging from small examples to real industrial case studies, e.g. [2, 7, 11].

For the verification experiment presented in this paper we used UPPAAL version 2.17 which improves previous versions specially with respect to its capabilities of deadlock analysis[4].

4 Formal Modelling of Jitter

Timed automata can be used as a convenient notation for formally specifying (and verifying) real-time properties like anchored and non-anchored jitter and skew. We illustrate this by means of a series of automata that generate streams, like video and sound streams. The availability of a frame is modelled by an output along channel s where we assume that such output is always possible (i.e. the environment is not imposing additional time constraints on the communication along s). Ideally the elapsed

[4] In the rest of the present paper we will often call "time-locks" those deadlocks which prevent time to pass (e.g. deadlocks involving committed locations), although in the UPPAAL terminology they are called simply deadlocks.

time between two successive frames is p. The automaton *Optimum Playout* (Figure 5, on the left) generates an optimal stream, without any jitter. After generating the first frame at an arbitrary time instant, it produces frames periodically with a period p.

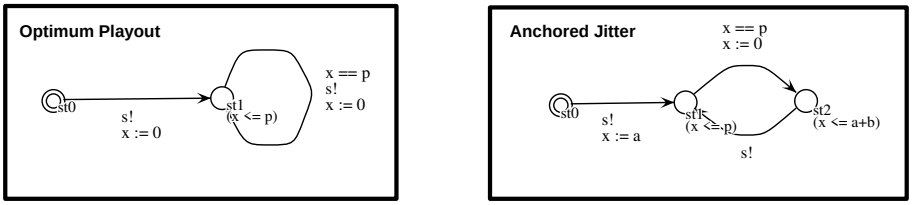
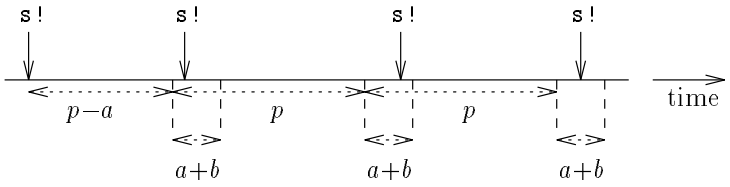


Figure 5: Timed automata for optimal playout and anchored jitter.

A stream exhibiting anchored jitter where frames are allowed to occur at earliest a time-units before the optimum presentation time, and at latest b time-units after this point in time, is generated by the automaton *Anchored Jitter*, see Figure 5 on the right. The stream of frames that it generates is:



The automaton presents a frame at some time instant in the indicated intervals of length $a+b$. This time instant is chosen non-deterministically. An automaton that generates a stream exhibiting non-anchored jitter is depicted in Figure 6 (left). Each frame (apart from the initial one) is presented within an interval $[p-a, p+b]$ of the presentation of the previous frame. Notice that for $a=b=0$ we obtain an automaton that is equivalent to the automaton generating the optimal stream.

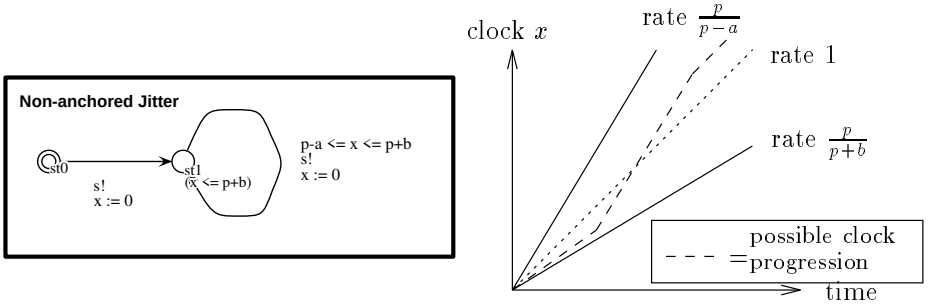


Figure 6: Timed automaton for non-anchored jitter and fluctuating clock rate.

In the optimal situation the automaton *Non-anchored Jitter* presents frames with a frequency of $\frac{1}{p}$ frames per time-unit; in the slowest case this frequency is $\frac{1}{p+b}$ and in the

fastest case $\frac{1}{p-a}$. One might consider that the period between two successive presentations is determined by a clock with fluctuating rate. This suggests an alternative specification of non-anchored jitter using so-called *linear hybrid automata*, timed automata in which clocks may proceed at different, but linearly dependent, rates. Consider the automaton *Optimal Playout* and adapt the rate of clock x such that it proceeds with a minimal rate of $\frac{p}{p+b}$ and a maximal rate of $\frac{p}{p-a}$. These rates are depicted in Figure 6 (right). While running, the clock may choose at any time instant any rate between these two values. If it always proceeds with rate 1 the clock proceeds as fast as time progresses, and the hybrid automaton boils down to the automaton *Optimal Playout*. UPPAAL supports the specification and verification of linear hybrid automata by using an algorithm that converts such automata into timed automata [16]. Indeed, if we apply this transformation on our hybrid automaton we obtain a timed automaton that is equivalent to the automaton *Non-anchored Jitter*.

Notice that in the above automata for anchored and non-anchored jitter, the exact point of time at which a frame is presented is completely non-deterministically determined. For several purposes it is of interest to quantify the probability of presentation at a certain time instant. For instance, consider anchored jitter where the probabilities of presentation at a certain time instant in the window $a+b$ is equal, i.e. uniformly distributed. Using *stochastic automata* [6] this can be specified as depicted in Figure 7 (left) where F is a deterministic distribution of p and G a uniform distribution in the interval $[0, a+b]$. For the sake of simplicity we do not make an exception for the initial presentation. By changing G other distributions can be used for quantifying the form of jitter. In a stochastic automaton clocks run backwards, and are initialised with a sample of an arbitrary probability distribution function. Clock expirations (i.e. a clock has reached value 0) can be used as guards. State invariants are absent: edges are taken as soon as they are enabled. A stream that exhibits non-anchored jitter with

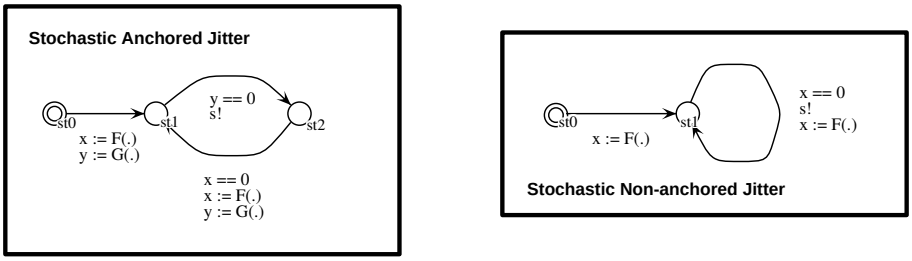


Figure 7: Stochastic automata for uniformly distributed anchored and non-anchored jitter.

a uniformly distributed probability is generated by the automaton in Figure 7 (right) where F is a uniform distribution in the interval $[p-a, p+b]$. Each frame is presented within a uniformly distributed interval $[p-a, p+b]$ of the presentation of the previous frame. The probability that, for instance, the delay between two successive frames is exactly p is $\frac{1}{a+b}$. Stochastic automata are not supported by UPPAAL, nevertheless

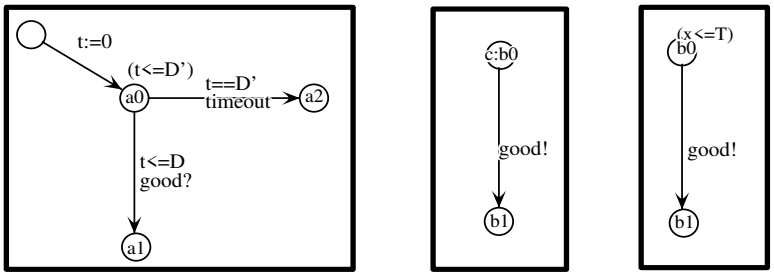


Figure 8: Bounded timeout

it should be clear from the above discussion that they are very useful for formally specifying basic concepts in the field of multimedia systems.

5 Formal Modeling of Timeout

In the scope of this paper we need two different kinds of timeout functionality. In the following we shall discuss and define them.

By a *Bounded* timeout we mean a device which, once activated, produces a *timeout* action at specified time D' (relative to device activation) if and only if a certain specified action *good* did not occur *by* time $D < D'$. Notice that this implies that if action *timeout* occurs, then it must occur at time D'; moreover, if action *good* occurs, then it can occur at any time from the timeout activation up to, and including, D. This is a *strong* timeout, in the terminology of [15].

In the context of this paper, it is assumed that if action *good* is enabled (i.e. can occur) before time D (from timeout activation time) it will occur by time D; actually, we further strengthen this assumption, by requiring that action *good* must be executed as soon as it is enabled, i.e. it is urgent. In the domain of media presentation this assumption is supported by the observation that if a frame is available it should be processed. It makes no sense to delay such a processing for no reason beyond the timeout deadline when the frame is available.

In Figure 8 (left) it is shown how this variant of timeout can be modelled in UPPAAL. Usually, such an automaton is embedded in a more complex one. Timeout activation is modelled by passing control to location $a0$, by means of an incoming further transition where the clock t is reset. The transition from $a0$ to $a1$ models the (in time) execution of the *good* action, while the other, from $a0$ to $a2$, models the signalling of the *timeout* action.

The fact that the timeout occurs exactly at time D', if it needs to occur, is modelled by a combination of the guard $t = D'$ at the transition labelled by *timeout*, and the invariant at location $a0$ that requires $t \leq D'$. The fact that *good* should happen at latest when t reaches the value D is expressed by the guard $t \leq D$ at the transition

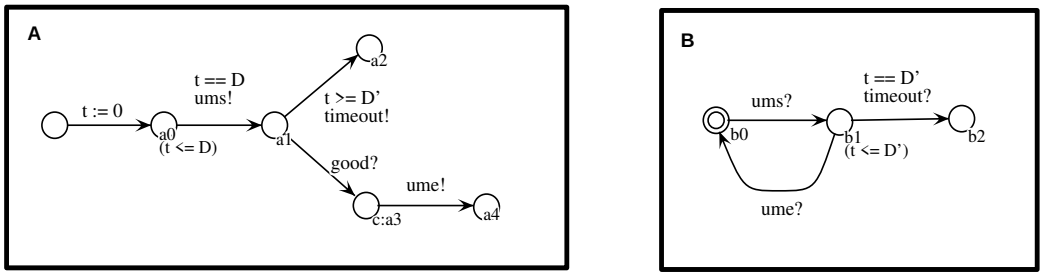


Figure 9: Precise timeout

labelled by *good*.

As stated above it is assumed that action *good* occurs as soon as it is enabled. It is easy to see in the figure that the timeout itself does not enforce *good* to be urgent since this is an assumption on its environment. This has to be guaranteed by other means, in the context in which the timeout is placed. As has been pointed out in section 3, UPPAAL provides three ways for expressing urgency; they are committed locations, urgent channels and through combinations of guards and location invariants. Unfortunately, the *good* action cannot be defined as an urgent channel because in the timeout construction it is guarded by $t \leq D$ and the use of guards for urgent channels is not allowed in UPPAAL. So the only way to make *good* urgent is to use committed locations or location invariants. These committed locations and location invariants will belong to automata that synchronize via *good* with (the automaton containing) the timeout (see Figure 8 (automata on right)).

It is important to point out here that special care is required in the use of such committed locations and location invariants since they can easily generate dead-/time-locks. In particular, it is worth pointing out here that once a *timeout* action has occurred, it is no longer possible for a *good* action to occur, which can result in dead-/time-locks.

A *Precise* timeout is essentially a Bounded timeout with the difference that action *good* must happen *at precisely* time D (and not *by* time D). Figure 9 shows how this variant of timeout can be modelled in UPPAAL. As we will discuss later, this model is only an approximation of a Precise timeout.

We first of all point out that in this case it makes no sense to model the urgency of the *good* action by means of committed locations or location invariants. In fact, in order to avoid time-lock, a committed state should be entered exactly when the *good* action should occur, which would nullify the use of the timeout. This would also be the case if a location invariant in the environment would be used; in fact, in order to be effective, it would require an upper bound equal to D (relative to timeout activation). Therefore we declare *good* as an urgent channel.

The timeout is modeled as two automata A and B and a clock t. The timeout is activated by passing control to location $a0$, in automaton A, by means of an incoming

transition where the clock t is reset, also making sure that automaton B is in its initial state. Automaton A will stay in location $a0$ until t reaches the value D. Exactly when $t = D$, this automaton makes a transition to location $a1$, from which both the *good* action and the *timeout* are possible. Action *timeout* is possible at any time not smaller than D'. In order to make it occur exactly when $t = D'$ we use automaton B; this automaton makes a transition to location $b1$ exactly when $t = D$ because of synchronization action $ums?$ paired with $ums!$ in the transition to $a1$ in A. It must leave location $b1$ at latest when $t = D'$, which is the time at which the transition to $b2$ can occur. This transition will execute action *timeout*! which will force transition to $a2$, labeled with *timeout*? to occur.

On the other hand, action *good* is allowed to occur at any time before the *timeout* occurs. Whenever this happens, we have to disable the timeout, in order to avoid a live-lock. This is done by means of making location $a3$ *committed*, which will force action $ume!$ of A to be executed together with action $ume?$ of B which brings it to its initial location.

A few considerations are now due. First of all, we have to point out that our UPPAAL model of the Precise timeout tolerates an occurrence of the *good* action at any time when $D \leq t \leq D'$, whereas we had required that such an action should be allowed *only* when $t = D$. Moreover, exactly when $t = D'$, both the *good* action and the *timeout* may occur, thus we have indeed modeled a *weak* timeout. The first problem has again to do with the fact that one is not allowed to associate a clock guard (like $t = D$) with a transition labeled by an action on an urgent channel (like *good*). A similar situation arises with invariants on locations which act as source for a transition with an urgent channel. This in turn implies that *good* can happen any time until the timeout expires, including when $t = D'$. In any case, if the *good* action is available when $D \leq t < D'$, then it is *guaranteed* to happen, and this is the maximum we can guarantee with this model of timeout. Thus, it is not possible to model a Precise timeout in UPPAAL [21]; probably, some notion of priority could help in these situations, but it is not provided by the tool.

6 Formal specification of the lip synchronisation protocol

In this section we give a formal specification in UPPAAL that follows as closely as possible the timed LOTOS specification given in [17] which covers the specification of the video and sound managers and the synchronizer (see Figure 2).

The full UPPAAL specification is shown in Figure 10, where the *Video Manager*, the *Sound Manager*, the *Video Watchdog* and the *Sound Watchdog* are modeled respectively by automata *VideoMgr*, *SoundMgr*, *VideoWdg* and *SoundWdg* (together with *UrgMon*). The *Synchronizer* is composed by automata *Synch*, *VideoSynch*, *SoundSynch*

and *SoundClock*. Finally, in order to perform verifications, we also need to model the "external environment", i.e. the incoming video and sound streams (*VideoStr* and *SoundStr*). In the following we shall briefly discuss these components.

The stream managers Both managers are triggered by the availability (at the presentation device) of a video or sound item respectively. This is modelled by the actions *savail* and *vavail*. The availability of a media item is immediately reported to the synchronizer via actions *sready* and *vready*. Immediately in this context means without delay and without interference of other actions. This is modelled in the managers by marking the locations *vm2* and *sm2* as committed. The managers must then wait for an indication from the controller (actually the watchdogs) that the media item is to be presented. This is modelled by the actions *vokk* and *sokk*. As soon as the indication has been obtained the presentation device must be given a signal to present the item. This is modelled by the internal actions *vpresent* and *spresent*. These actions are left internal because the presentation device itself is not further specified[5]

The watchdog timers Each watchdog timer ensures that the time between two consequent presentations of media items of the same kind is between certain bounds. If a media item is too late for presentation the watchdog timer has to give an error signal. We first consider the video watchdog. Initially it waits for the first presentation of a video frame, which is indicated by the *vok* action. This action precedes the *vpresent* but it is guaranteed that no time passes between the *vok* and the presentation of the video frame. At the moment the first *vok* is observed, a clock *t4* is started and the action *vokk* is issued to the video manager without any delay. The combination of *vok* and *vokk* makes the synchronisation between three automata possible, namely the automata *VideoMgr*, *VideoWdg* and *VideoSynch*. The two actions *vok* and *vokk* can therefore be considered as one atomic action[6]. The *VideoWdg* has to guarantee that the next video frame is presented between 35 ms and 45 ms after the previous one. Therefore the transition labelled by *vok* leaving location *vw3* is guarded by $t4 >= 35$ and $t4 <= 45$. When *vok* occurs the clock *t4* is reset to zero. Immediately after *vok* there is a committed transition labelled by *vokk* back to location *vw3* to start a new timeout session. If *vok* does not occur before 45 ms pass, a *vlate* error is given at time $t4 == 46$. Note that *VideoWdg* is modelled as a slight variation of a repeated Bounded timeout (see section 5).

The *SoundWdg* is a bit more complicated. Essentially it has to take care that a

[5] In [17] there is an additional action *presented* that is performed by the presentation device to mark the end of the presentation of a media item. We have omitted this action because in the timed LOTOS specification this action was apparently assumed to occur always before the next media item becomes available and therefore cannot create further complications for the protocol.

[6] Although interleaving is allowed in this case

sound frame is presented exactly at every 30 ms. If a sound frame is too late for presentation it should generate an error indicating that the sound is late 1 ms after its original presentation time. The initial part of *SoundWdg* is similar to that of *VideoWdg*. When the first *sok* is observed a timer *t3* is started and synchronisation on this action with the *SoundMgr* is established via *sokk*.

The repeated timeout construction is of kind Precise, as discussed in section 5. The *SoundWdg* waits in location *sw3* until 30 ms have passed since the last occurrence of a sound frame. At that time it notifies the urgency monitor *UrgMon* by means of action *ums* and it resets clock *t3* to zero. At this point an *sok* can happen urgently (*sok* is defined as an urgent channel) or, if *sok* is not available, an *slate* error is generated 1 ms later. The construction with *UrgMon* is needed to guarantee that *slate* happens urgently.

If the *sok* happens in time, *UrgMon* is immediately notified by *ume* about this fact and the *sokk* action is generated to model the multi part synchronisation with the *SoundMgr*.

The synchronizer The synchronizer *Synch* is activated by *vready* or *sready*. Depending on which of these actions occurs first it generates a *vok* or an *sok* and after that it starts three automata in parallel. The initial part of these automata is different and depends only on whether a video frame or a sound frame has been received first. To start the automata in the right way their initialisation is synchronized on special actions that do not occur in the original Timed LOTOS specification. In this way we model the parallel composition operator that is available in LOTOS but not as such in UPPAAL. The names of the special actions are a shorthand of the following.

- *std* (*sti*) initialises the *SoundClock* in case a video (sound) frame arrived first.
- *sv1* (*sv0*) initialises the *VideoSynch* in case a video (sound) frame arrives first.
- *ss0* (*ss1*) initialises the *SoundSynch* in case a video (sound) frame arrives first.

Note that all the locations except the initial location of the *Synch* are committed. This is necessary to model that the three automata start *at the same time* in parallel immediately after the first *vok* or *sok* action.

The sound clock
The *SoundClock* is a discrete clock that ticks with units of 1 ms. It is started at the moment that the first sound frame has arrived and is presented. This clock serves as a reference time to compute the amount of skew that the video stream may have with respect to the sound.

If a sound frame arrives as first frame the clock is started via *sti* and forced to perform a transition every 1 ms. During this transition a variable *vmins* is updated that keeps a record of the amount of skew between the sound and the video stream.

This variable is called *vmins* because of its direct relation to the original Timed LOTOS specification in which the time of the sound presentations was recorded in one variable (s-time) and the ideal time of the video presentation in another variable (v-time). The skew was calculated by subtraction of s-time from v-time. Notice that, at the time at which a video frame arrives, s-time corresponds to the arrival time of such a frame.

If a video frame arrives first, the *SoundClock* automaton is started by means of the *std* action. In this way the clock ticks start only after synchronization on action *sclock* has indicated the arrival of the first sound frame.

The sound synchronizer Also the sound synchroniser can start in two different ways. If a sound frame arrives first it directly starts its repeating behaviour via synchronisation on action *ss1*. The repeating part of the behaviour is very simple and consists only of receiving an *sready* action after which an *sok* is generated. Note that the *sok* action is defined as an urgent channel in the configuration in order to let *sok* happen as soon as possible.

If a video frame arrives first the sound synchroniser starts by checking whether a sound frame arrives within 15 ms of the initial video frame. This is part of the requirement for lip synchronisation. If the sound frame does not arrive in time a synchronisation error is generated 16 ms after the start of the sound synchroniser (Bounded timeout). If the sound frame arrives within 15 ms, an *sok* action is generated immediately, the sound clock is started via the *sclock* action and the automaton starts its repeating behaviour.

The video synchronizer The video synchroniser is the most complex process of the lip synchronisation protocol. If a video frame arrives first it starts the repeating part of its behaviour via synchronisation on action *sv1*. From that point on the video synchroniser essentially checks the lip synchronisation requirement and generates an error if there is too much skew between the video and the sound stream.

In every cycle *VideoSynch* waits for a *vready* action. When it receives a *vready* it resets clock *t1* to zero and goes to state *v03* where it checks the lip synchronisation requirement immediately (due to the invariant $t1 <= 0$). Now there are three possibilities:

1. The video presentation is more than 150 ms later than the corresponding sound presentation. This situation is characterised by the guard $vmins < -150$. In this case a synchronisation error is produced.

2. The video is more than 15 ms too early with respect to the corresponding sound presentation. In this case the video presentation can be delayed. This situation is modelled by the guard $vmins > 15$. It leads to a state in which the video synchroniser is forced to wait 1 ms and then repeats the checking of the lip synchronisation requirement.

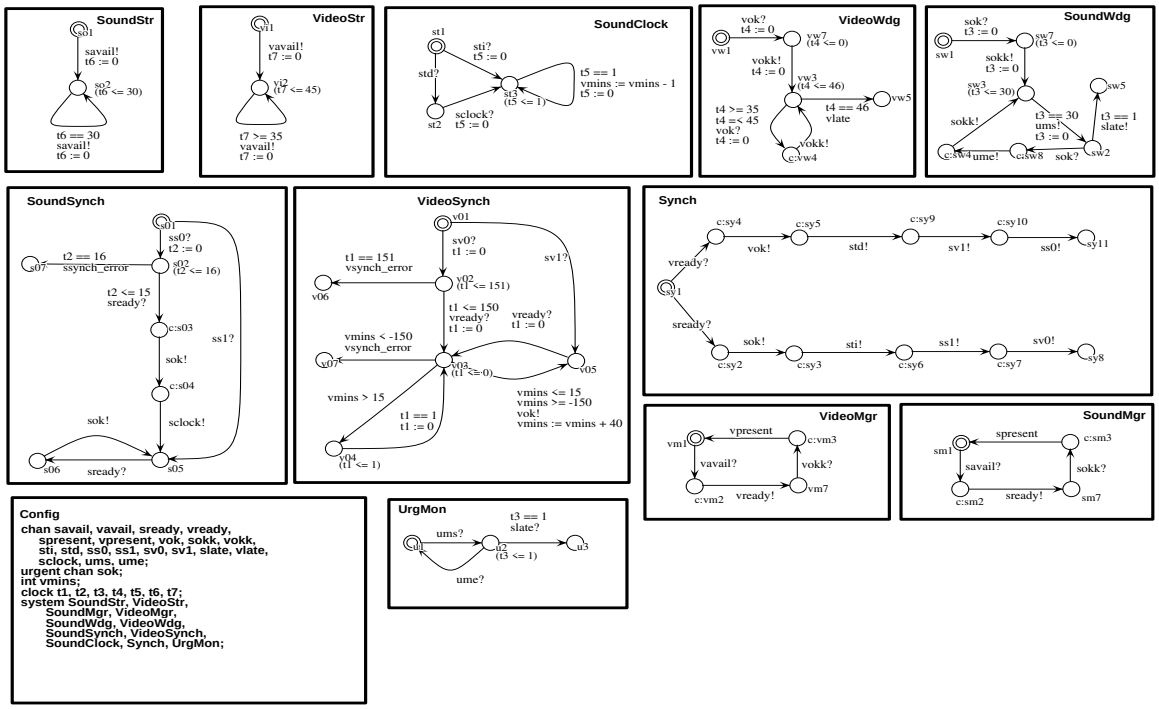


Figure 10: Lip synchronisation protocol

3. The video presentation is sufficiently in synchronisation with the sound presentation. This situation is characterised by the guard $vmins <= 15$ and $vmins >= -150$. In this case a *vok* is generated immediately and the variable $vmins$ is updated.

If a sound frame arrives first only the initial behaviour of the video synchroniser is different. In this case it checks if the first video frame arrives within 150 ms of the first sound frame. If the video is too late a synchronisation error is generated. If the video frame is in time it starts its repeating behaviour by checking the lip synchronisation requirement.

The media streams Since the informal specification does not describe any assumptions on the streams, we can in principle model them as we like. Unfortunately it soon becomes clear that the protocol is not able to deal with all possible streams. The models of the media streams we used are further described in the section on verification.

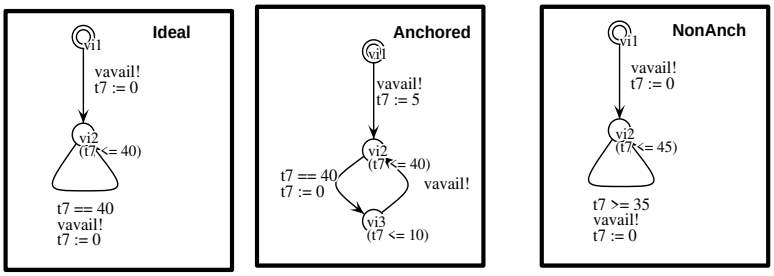


Figure 11: Three variants of video stream behaviour

7 Verification

7.1 Verified properties

In all the specifications of the lip-synch protocol found in the literature little is said about the assumptions that were made on the behaviour of the media streams from the receiver point of view. In our verification we investigated several different behaviours of the media streams.

For any model of the sound stream that does not let frames arrive every 30 ms the lip-synch protocol does not behave properly. It seems clear that the lip-synch protocol has been designed assuming a perfect behaviour of the arriving sound frames. We do not present here the specific verification results we have got on this aspect. We rather make the explicit assumption that the sound stream does not show any perturbation.

Instead of verifying properties that have been reported in work in the literature on lip-synch, such as proving that a sound frame is presented every 30 ms [9], we take such basic properties for granted and we explore possible problems caused by jitter of the video stream.

We investigated the results for three kinds of video stream behaviour:

- An "ideal" video stream that delivers a frame every 40 ms.
- A video stream with "anchored jitter" with rate of 40 ms and variation of $\pm$ 5 ms.
- A video stream with "non-anchored jitter" where the variability between each two consequent frames is minimally 35 ms and maximally 45 ms.

These automata are instantiations of the automata for jitter we have shown in Sect. 4 and are shown in figure 11.

Each video stream behaviour has been investigated in the situation in which the start of each stream was left unspecified and the situation in which both streams start at the same time.

In the verification we did a reachability analysis of the error conditions. The results for each situation have been obtained by checking the reachability property on the model consisting of the lip-synch specification and a variant of the video stream. The reachability properties are all of the form

$$E <> A.l \; and \; not(B_1.l_1 \; or \; ... \; or \; B_n.l_n)$$

i.e. does there exist a path in which eventually the control of automaton A is in location l and the control of other automata B_i is not in certain other locations l_i. In these properties location l is the location that indicates the error the reachability of which we are checking. The second part ensures that the other automata did not reach another error location. In this way we know that the error we are checking for did not occur as a consequence of other errors. It is worth recalling here that all error states of our automata are sink states. In the lip-synch protocol, the following error locations have been modelled:

- Initial sound synchronisation error in the *SoundSynch* (location $s07$)
- Initial video synchronisation error in the *VideoSynch* (location $v06$)
- Video synchronisation error in the *VideoSynch* (location $v07$)
- Video late error in the *VideoWdg* (location $vw5$)
- Sound late error in the *SoundWdg* (location $sw5$)

7.2 Results

We ran a first verification suite on a SUN Ultra SPARC 143, running SUN-OS 5.5.1 with 128 Megabytes of RAM. We were unable to successfully complete all verifications because of resource limitations, especially in terms of disk space needed for diagnostic files. Meanwhile we found that UPPAAL 2.17 was about five times faster when running on a PC with a AMD K6 processor at 200Mhz with 64 Megabytes RAM and with the Red Hat Linux 5.0 operating system, so we used such a PC for all subsequent verifications. Moreover, we reduced the state space of the model by marking all error locations as committed forcing a time-lock whenever control reaches any such location.

Figure 12 gives the result of verification of the lip-synch protocol for the various reachability properties when there may be an initial delay between the streams. The leftmost column lists which kind of reachability error has been checked. For each type of video stream behaviour the result of the reachability check and the C.P.U. time in seconds are reported. The numbers between brackets at a 'True' in the table give the least number of time units (ms) that are needed to reach the error.

From the table it is clear that initial out_of_synchronisation errors for both the video and the sound can always occur. This is explained by the fact that the time between the

	possible initial delay between streams					
Property	Ideal Video		Anchored Video		Non-anchored Video	
Init Sound Synch err	True (16)	0.08	True (16)	0.07	True (16)	0.05
Init Video Synch err	True (151)	145.47	True (151)	6479.72	True (151)	246.42
Video Synch err	False	291.70	True (191)	16143.67	True (191)	421.39
Video Late	False	291.72	True (81)	410.45	False	2638.93
Sound Late	False	291.69	False	32899.19	False	2638.36
Deadlocks	1		1		1	

Figure 12: Verification results for streams with initial relative delay

first video and the first sound frame can be arbitrarily long. The error occurs exactly when the maximal delay has passed, so at 16 ms and at 151 ms respectively. If these initial errors do not occur, the ideal video stream cannot go out of synchronisation with the sound stream. The model checker performs a complete search in about 292 seconds.

The anchored and non-anchored streams *can* go out_of_synchronisation. The anchored stream can wait to start sending frames until the latest time that does not create an initial video synchronisation error. It's delay w.r.t. sound is then already large. When the next video frame arrives as late as possible given the jitter, it creates an out of synchronisation error. The non-anchored stream can go out of synchronisation in a rather similar way.

Video frames can arrive late only in the case of anchored jitter. This is explained by the fact that the time between two consecutive frames in the stream with anchored jitter is maximally 50 ms. This is 5 ms more than *VideoWdg* allows. Sound frames can never be late. This is of course because we modelled the sound stream as an ideal stream.

When both streams are forced to start at the same time the results of the verification are rather different, as shown in Figure 13. The ideal video stream does not lead to any error or deadlock. This is what we would indeed expect.

The anchored stream can lead to a late arrival of a video frame. This is for the same reason as in the case when initial delay between the two streams is allowed.

The non-anchored stream can lead to an out of synchronisation error because of the possible cumulation of delay w.r.t. the sound stream (skew).

7.3 Deadlocks

The last rows of both tables indicate whether the verifier reported any deadlocks which were not caused by reaching an error state. When the video and the sound start at the same time no deadlocks were reported. When they start independently one deadlock

	NO initial delay between streams					
Property	Ideal Video		Anchored Video		Non-anchored Video	
Init Sound Synch err	False	1.060	False	23.150	False	2641.600
Init Video Synch err	False	1.080	False	23.130	False	2644.570
Video Synch err	False	1.080	False	23.180	True (1031)	2483.120
Video Late	False	1.060	True (81)	3.97	False	2641.560
Sound Late	False	1.080	False	23.150	False	2644.250
Deadlocks	None		None		None	

Figure 13: Verification results for streams without relative initial delay

was reported for every type of video stream. These deadlocks are very similar. We start by discussing the deadlocks that have been reported and continue with the discussion of some problems we found incidentally.

The deadlocks that have been reported by UPPAAL and that were not due to reaching an error location were all related to how the timeout modelled in *VideoSynch* at location $v02$. In each specification, such a deadlock occurs when the first sound frame has been received and the first video frame arrives between (but not including) 150 ms and 151 ms after the sound frame. In that situation the *VideoMgr* synchronises on $vavail$ and has to do a $vready$ immediately due to the committed location $vm2$. This $vready$ cannot be performed because $t1$ is beyond 150 ms in location $v02$ of *VideoSynch*. This leads to a time-lock. This kind of time-lock has already been highlighted in Section 5. In the original lip-synchronisation protocol described in [17] this problem could not occur because a discrete time model was used. This time model implicitely presupposes that frames arrive only at discrete points in time so for example only at precise ticks of a clock. This assumption was not made explicit in the problem description of the lip-synchronisation.

We would have expected UPPAAL to report a similar deadlock in *SoundSynch* but even a full state space search did not reveal it. We think that to fully explain this requires further research.

With the anchored video the verifier did not report any further deadlock either, but by means of the simulator we have found a timelock just after a few transitions from the starting state. This timelock occurs when a video frame arrives as late as allowed by the loop in the specification of the video stream and the next video frame arrives as early as possible. The time between the arrival of these two frames is 30 ms. The *VideoSynch* needs to synchronize urgently with the *VideoWdg* on the vok action, but the *VideoWdg* is at that point still waiting until at least 35 ms have passed since the last video frame.

Also in the case of non-anchored jitter no further deadlocks were reported. However, a small change in the *VideoStr* that replaces the invariant by its strict version $t7 < 45$

leads to a timelock situation. This timelock is a very interesting one because it reveals another, quite hidden, problem of the lip-synch specification. The timelock occurs when *VideoStr* has to synchronize on *vavail* with *VideoMgr* just before 45 ms passed, and *VideoSynch* is at location $v04$ because the video was too early with respect to sound (i.e. $vmins > 15$). In this situation the above synchronization on *vavail* cannot take place because *VideoMgr* is at location $vm7$ waiting to synchronize on *vokk* with *VideoWdg* due to the video being early w.r.t. sound. In order to enable the synchronisation on *vavail* time must pass because of the guard $(t1 == 1)$ on the outgoing transition at location $v04$ in *VideoSynch*. Due to this forced delay, the invariant $t7 < 45$ at location $vi2$ of *VideoStr* cannot be satisfied, thus leading to the timelock. In the non-strict version $(t7 \leq 45)$ this timelock is avoided in a curious way. The synchronization on *vavail* between *VideoStr* and *VideoMgr* is *delayed* until $t7 == 45$ so that *VideoSynch* can leave location $v04$ by pure time passing and subsequently synchronize on *vok* with *VideoWdg*. This enables the synchronization on *vokk* between *VideoWdg* and *VideoMgr* and *vpresent* to occur at the *VideoMgr*. Since the complete sequence of transitions, after *VideoSynch* has left location $v04$, occurs without consuming time because of the concatenation of committed locations, the synchronization on *vavail* between *VideoStr* and *VideoMgr* can take place as well.

This aspect of the lip-synch is not satisfactory because in reality it is unlikely that the arrival of a frame can be postponed until a proper time. The arrival of a frame is determined by the environment in which the lip-synch protocol works rather than by the protocol itself. It is easy to see that there is a period in which both video and sound managers are not available to receive any frame, namely when they are waiting for a *vokk* and a *sokk* respectively. Notice also that the next frame can be received only after a *vokk* (resp. *sokk*) for the previous frame has been communicated.

8 Conclusions and Related Work

We have specified and verified a lip-sync algorithm in UPPAAL. Specifications of this algorithm have been made previously in a number of different formalisms, [18, 3, 17, 9]. We have particularly followed the timed LOTOS specification to be found in [17]. We found it interesting to investigate how several typical multi-media concepts, like jitter, drift and skew can be formally specified and analyzed.

Our verification has identified a number of interesting issues with the algorithm, of which, two of the most important are:

- The last column of figure 13 indicates that with non-anchored jitter, which was the variety of jitter the specification was defined for, and both streams starting together, lip-sync can only be guaranteed for *just over one second* (1031 ms). This is clearly quite a low figure. However of course, if we reduced the amount of pertubation allowed on the video stream then this length of time would increase. This points to one of the strengths of the form of verification we have considered:

we can derive bounds on the performance of components of the system (here the video stream) under which the system will behave satisfactorily.

- In addition to only guaranteeing lip-sync for a short period of time, our verification work has also highlighted some concrete problems in the algorithm. In particular, we have shown that with all types of video streams we defined a timelock can be reached in which none of the components is in a prescribed error state. Some of these timelocks have been found automatically, others where found by simulation. Some timelocks appeared because we used a dense time model to describe an algorithm previously specified in a discrete time model. Other deadlocks were related to the fact that the assumptions on the behaviour of the media streams have not been made explicit in the original problem description.

Another limitation of the lip-synch algorithm is that it does not handle buffering, i.e. the possibility that the presentation device smooths out synchronization errors by buffering packets before playing them. Adding buffering would increase the capability of the algorithm to handle pertubated sound and video streams and in addition, would enable a number of the problems with the existing algorithm to be resolved. We are currently investigating the possibility to add such buffering.

The work reported here has also enabled us to evaluate the UPPAAL tool in the context of a non-trivial specification and verification scenario. Our experience with UPPAAL (especially the most recent version of the tool) has generally been positive. Nonetheless we can point to some limitations:-

- **Class of Properties Checked.** A known limitation of the tool is that it only performs reachability analysis and thus, only checks a small subset of the full class of timed temporal logic formulae. A strategy for checking *bounded* liveness properties using test automata has been proposed and we have investigated such a strategy in verifying latency properties [5]. However, the strategy is not implemented yet and thus, has to be performed by hand.

- **Timelocks.** A major aspect of the verification of time sensitive systems is to check that states cannot be reached in which the passage of time is blocked. Such states often represent major specification errors. For example, our analysis has identified situations in which a timelock can arise without being in a prescribed error location. However, we have not identified all these states through direct verification for timelock freedom. In particular, one of them was not revealed by the model checker, but rather through simulation. Why some timelocks have not been reported during full state space search of the specification is not clear at this moment and requires further research. It would be interesting to repeat our experiment with some other tool like KRONOS [8]. KRONOS accepts a richer set of temporal logic formulae, including "unbounded" liveness properties. Freedom from timelocks can be coded up as an "unbounded" liveness property. KRONOS

also offers the possibility to use a clock reduction algorithm, which automatically reduces the number of clocks used. This could be very effective in the context of the lip-sync specification, which contains many clocks.

- **Low Level Notation.** Timed automata can also be criticised on the grounds that they are a relatively low level notation. For example, timeout operators and watchdog timers have to be "hand wired". Also, our investigation in section 5 suggests that certain forms of "strong" timeout behaviour cannot be easily described in the UPPAAL notation and some forms can be only approximated. This can easily lead to the introduction of timelocks. It would be nice to have a set of generic high level operators for timed specification, which could be mapped down to timed automata. Furthermore, the timed automata notation only allows one level of parallel composition, i.e. component automata cannot themselves contain parallel compositions. This leaves a question mark over the scalability of the notation.

Acknowledgements

We would like to thank Wang Yi and Paul Pettersson, of Uppsala Univeristy, for advice on UPPAAL and Stavros Tripakis of VERIMAG-SPECTRE for fruitful discussions on symbolic model checking. In addition, David Duke, of the University of York, was involved in some preliminary work on verifying the lip-sync algorithm.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, (126):183–235, 1994.

[2] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using uppaal. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification*, LNCS 1102, pages 244–256, New Brunswick, New Jersey, USA, July 1996.

[3] G.S. Blair, L. Blair, H. Bowman, and A. Chetwynd. *Formal Specification of Distributed Multimedia Systems.* University College London Press, September 1997.

[4] H. Bowman, L. Blair, G.S. Blair, and A. Chetwynd. A formal description technique supporting expression of quality of service and media synchronisation. In *Multimedia Transport and Teleservices, COST 237 Workshop, LNCS 882.* Springer-Verlag, 1994.

[5] H. Bowman, G. Faconti, and M. Massink. Specification and verification of media constraints using UPPAAL. In *Accepted for publication in the Proceedings of the 5th Eurographics Workshop on the Design, Specification and Verification of Interactive Systems, DSV-IS 98, Abingdon, UK.* Springer-Verlag, 1998.

[6] P.R. D'Argenio, J.-P. Katoen and E. Brinksma. An algebraic approach to the specification of stochastic systems (extended abstract). In D. Gries and W.-P. de Roever, editors, *Proceedings IFIP Working Conference on Programming Concepts and Methods*, 22 pages. New York, USA. Chapman & Hall, 1998.

[7] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1217, pages 416–431, Enschede, The Netherlands, April 1997.

[8] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, LNCS 1066*. Springer-Verlag, 1996.

[9] A. Feyzi Ates, M. Bilgic, S. Saito, and B. Sarikaya. Using timed CSP for specification, verification and simulation of multimedia synchronization. *IEEE Journal on Selected Area in Communications*, 14:126–137, 1996.

[10] S. Fischer and S. Leue. Formal methods for broadband and multimedia systems. *Computer Networks and ISDN Systems, Special Issue on Trends in Formal Description Techniques and their Applications, to appear*, 1998.

[11] Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou. Modelling and analysis of a collision avoidance protocol using spin and uppaal. In *Proceedings of the 2nd SPIN Workshop*, Rutgers University, New Jersey, USA, August 1996.

[12] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2), October 1997.

[13] P. F. Linington. RM-ODP: The Architecture. In K. Raymond and L. Armstrong, editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 15–33, Brisbane, Australia, February 1995. Chapman and Hall.

[14] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[15] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In J de Bakker, C. Huizing, C. de Roever, and G. Rozenberg, editors, *Real-Time: Theory and Practice. REX Workshop*, volume 600 of *LNCS*, pages 526–548. Springer-Verlag, 1991.

[16] A. Olivero, J.Sifakis, and S.Yovine. Using abstraction techniques for the verification of linear hybrid systems. In *CAV'94*, volume 818 of *LNCS*, pages 81–94. Springer-Verlag, 1994.

[17] T. Regan. Multimedia in temporal LOTOS: A lip synchronisation algorithm. In *PSTV XIII, 13th Protocol Specification, Testing and Verification*. North-Holland, 1993.

[18] J-B Stefani, L. Hazard, and F. Horn. Computational model for distributed multimedia applications based on a synchronous programming language. *Computer Communications (Special Issue on FDTs)*, 15(2), 1992.

[19] R. Steinmetz. Human perception of jitter and media synchronization. *IEEE Journal on Selected Areas in Communications*, 14(1):61–72, 1996.

[20] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint solving. In *Proceedings of the 7th International Conference on Formal Description Techniques*, Berne, Switzerland, 4-7 October 1994.

[21] Wang Yi. Personal Communication.