Formal Aspects of Computing (1998) 10: 550–575

Formal Aspects of Computing

Automatic Verification of a Lip-Synchronisation Protocol Using UPPAAL

H. Bowman[1], G. Faconti[2], J.-P. Katoen[3], D. Latella[2] and M. Massink[4]

[1] Computing Laboratories, University of Kent at Canterbury, Kent, UK;
[2] CNR-Istituto CNUCE, Pisa, Italy;
[3] Informatik VII, U. of Erlangen-Nürnberg, Erlangen, Germany;
[4] Dept. of Computer Science, University of York, York, UK

Keywords: Model checking; Lip synchronisation; Specification; Timed automata; UPPAAL; Verification

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

1. Introduction

It is now well recognised that the next generation of distributed systems will be distributed *multi-media* systems, supporting multi-media applications such as video conferencing. Importantly though, multi-media imposes a number of new requirements on distributed computing, not least of which is the need to ensure "timely" transmission and presentation of multi-media data, e.g. ensuring that the end-to-end timing delay between transmitting and presenting video frames

Correspondence and offprint requests to: D. Latella, CNR-Istituto CNUCE, Via S. Maria 36, 56126 Pisa, Italy. E-mail: d.latella@cnuce.cnr.it

stays within acceptable bounds. Consequently, there is significant interest in how to determine that multi-media systems satisfy their real-time requirements (which, in the distributed multi-media systems field, are typically categorised as *Quality of Service (QoS)* properties).

Furthermore, it would be advantageous if the real-time properties of systems could be analysed more thoroughly during the early stages of system development. This could reduce development costs significantly. Formal specification and verification offers great potential in this respect.

Consequently, a number of researchers have considered techniques for the specification [BBB97, FiL98, Reg93, FBS96] and verification [BFM98] of multi-media systems. One contribution of this body of work is to identify a number of canonical examples of multi-media systems, e.g. a multi-media stream and a lip-synchronisation protocol (LSP) [BBB97]. The latter is particularly important as it offers a non-trivial example of real-time synchronisation between continuous media. It incorporates several fundamental notions of multi-media synchronisation, but it is still simple enough to be dealt with in full detail. In fact, the LSP has become a standard example in the field of formal approaches to multi-media systems design and several papers on the application of formal methods to this protocol can be found in the literature.

The LSP was first described in the synchronous language Esterel [SHH92]. Then specifications in a number of different formalisms were presented, e.g. in a timed LOTOS [Reg93], in a dual language approach, LOTOS/QTL, [BBB94, BBB97] and in timed CSP [FBS96]. Typically, these specifications describe the protocol in their chosen formalism and then postulate that it satisfies certain timing requirements. However apart from [FBS96], 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 LSP using the *real-time model checker* UPPAAL [LPY97]. 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 [FBS96].

This paper has three main goals. As discussed above, one of them is in itself the application of automatic formal verification to an example for which several formal specifications exist but essentially no formal verification has yet been reported. Delegating the verification to an automatic tool has made it possible to analyse the behaviour of the protocol in a number of different contexts. We particularly investigate the effect of, firstly, varying the type of sound and video streams and, secondly, changing the parameters embedded in these streams. We investigate for how long the protocol can maintain lip-synchronisation and whether it signals a 'proper' error when the synchronisation requirements are not met.

This by itself constitutes a second goal or result of this paper. In particular, our verification study shows that for certain sound and video streams the protocol can deadlock without reaching any of the prescribed error states.

The third goal is to report on our experience with UPPAAL for performing real-time model checking of a non-trivial example in the area of multi-media. In this respect, we found it rather easy and helpful to model key concepts like *jitter* and *skew* by means of the timed automata notation offered by UPPAAL while, surprisingly, we found problems in modelling other important notions like

time-outs. Furthermore, we had some problems also with respect to deadlock detection.

The paper is structured as follows. Section 2 introduces the lip-synchronisation 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 protocol. 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 multi-media 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 multi-media 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 [Lin95]).
- *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 between the generation of frames and their presentation becomes too large the sense of simultaneous interaction will be lost. Typical quality of service properties include: *end-to-end latency* between the generation of frames and their presentation (which we simply call *latency*), *throughput*, i.e. the rate at which frames are presented and *jitter*, which concerns the variability of delay, we consider it further in Section 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 multi-media stream with associated quality of service parameters was presented in [BFM98]. 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 been used inconsistently in the literature.

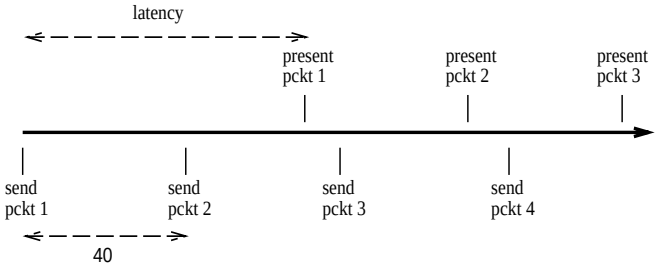


Fig. 1. An optimum playout of frames.

2.2.1. *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 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 frames 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 incur a drift relative to the optimum; +4 for the second frame, +8 for the third, +12 for the fourth and so on.

The anchored jitter interpretation appears frequently in the multi-media literature, however, much of the previous work on lip synchronisation has interpreted jitter in a non-anchored fashion [BBB97, Ste96].

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

2.2.2. Skew

In line with the terminology in [Ste96] we use the term skew to refer to the time difference between related audio and video frames. Thus, while jitter is an intra-stream measure, skew is an inter-stream measure. It categorises the degree to which the two streams are out of synchronisation. 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 [Ste96], 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 protocol we consider here is such a scenario.

Importantly though, re-synchronisation 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 [Ste96] 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 over-specify), the LSP accommodates certain out of synchronisation levels.

The basic system configuration that we consider is shown in Fig. 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 synchronised to an acceptable degree.

The protocol consists of a number of components: *sound* and *video* managers and a *controller*. When a sound frame 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 frame can be presented. The *Video Manager* has a corresponding behaviour. The *Controller* contains the body of the lip-synchronisation protocol. It receives *sready*s (respectively *vready*s) from the *Sound* (respectively *Video*) *Manager*, indicating that a *Sound* (respectively *Video*) frame is ready to be played. The *Controller* then evaluates if and when it is appropriate to play the particular frame. It either returns an *sok* (respectively *vok*) 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 [Reg93, SHH92][2].

[2] According to [Ste96] different figures should be used in practice. Although these figures are important they do not affect the essence of the LSP.

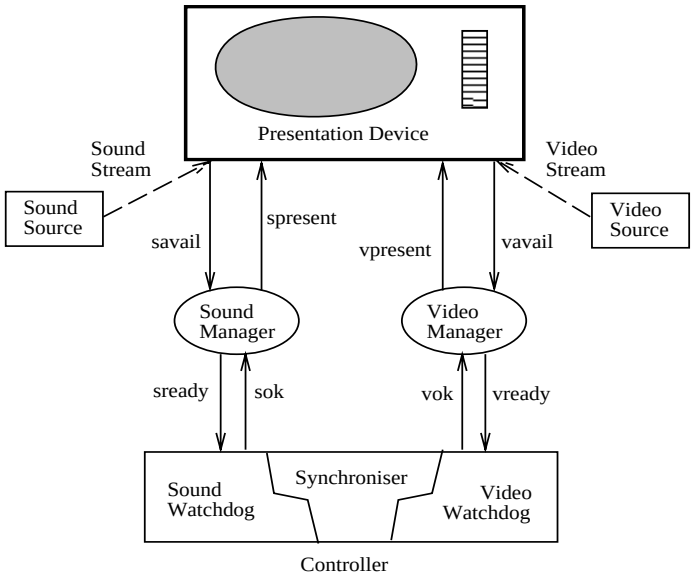


Fig. 2. Basic structure of the lip-synchronisation system.

- The granularity of time is a millisecond[3].
- A sound frame must be presented every 30ms (each sound frame contains 400 samples of 12kHz sampled digital sound). No jitter is allowed on the sound.
- In the optimum, a video frame 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.

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

3. Introduction to UPPAAL

UPPAAL is a tool-suite for the specification and automatic verification of real-time systems [LPY97]. It has been developed at BRICS in Denmark and at Uppsala University in Sweden. In UPPAAL (version 2.17) a real-time system is modelled

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

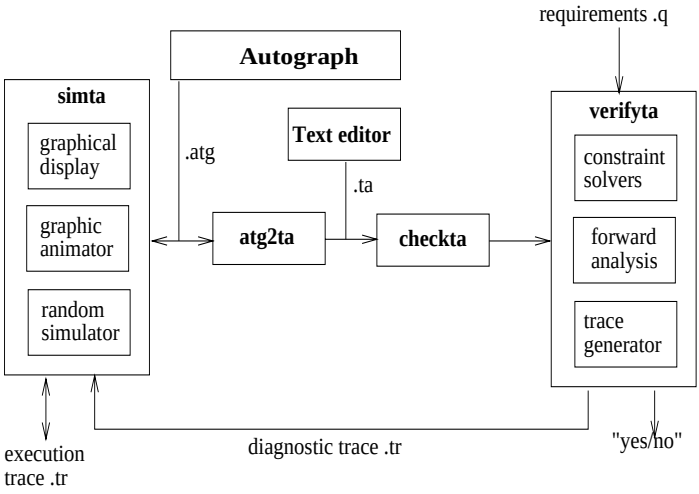


Fig. 3. Overview of UPPAAL tool suite.

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 Fig. 3 an overview is given of the different components of the UPPAAL tool and their relation.

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 either 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 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 clock and integer variables, the communication channels and the composition of the network are defined.

3.1.1. Transitions

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 [Mil89]. Given channel name a, $a!$ and $a?$ denote complementary actions corresponding to *sending* respectively *receiving* on channel a. Absence of a synchronisation action is interpreted as an internal action similar to the role of τ-actions in CCS.

3.1.2. Locations

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.

3.1.3. Urgent Channels

Communication channels can be declared to be *urgent*. When a channel is urgent, no timing constraints can be defined on the edges labelled by that channel and no invariant can be defined on the location from which that edges leave. A synchronisation on urgent channels has to happen as soon as possible, i.e. without delay, but interleaving of other actions is allowed if this does not cause any delay.

3.1.4. Interpretation

The interpretation of a UPPAAL model is in terms of states and transitions. 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:

- 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.

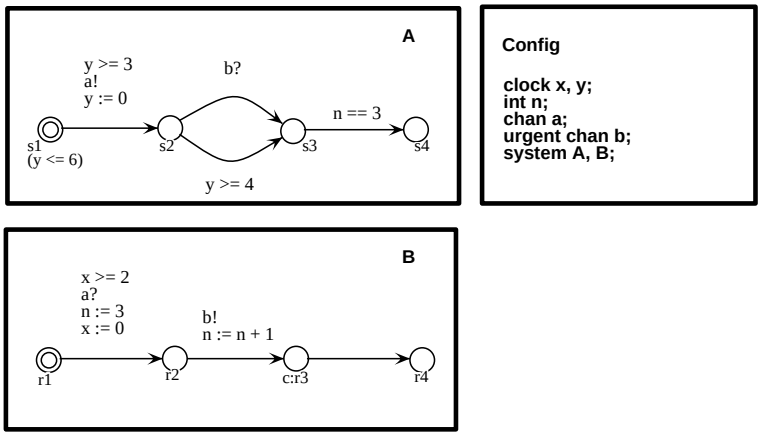


Fig. 4. Example of an UPPAAL specification

3.1.5. An Example

An example of an UPPAAL specification is given in Fig. 4. The transition between *s1* and *s2* can only be taken when the value of clock y is at least 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 at latest when 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 $y \geq 4$ 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 of 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

3.2.1. Regions

The future behaviour of a network of timed automata is fully characterised 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 [AlD94]. 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, the time domain can be discretised. In addition the state space of a model can be partitioned into finitely many regions because in every model there is a maximum integer with which a clock is compared. 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 [YPD94].

3.2.2. Logic for Specifying Requirements

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.

3.2.3. Data Types

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. (This can be mimicked by means of shared variables.) 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. [BGK96, DKR97, JLS96].

3.2.4. UPPAAL *Version Used*

For the verification experiment presented in this paper we used UPPAAL version 2.17 which improves previous versions, especially with respect to its capabilities for 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 time between two successive frames is p.

[4] In the rest of the paper we will often use the term *time-locks* for situations in which time is prevented to advance (e.g. due to the interplay of committed locations), although in the UPPAAL terminology they are simply called deadlocks.

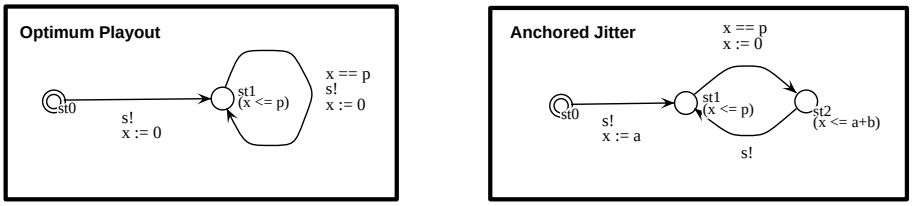


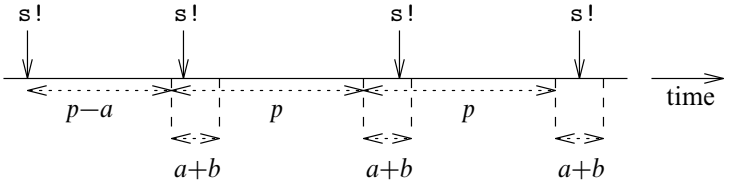
Fig. 5. Timed automata for optimum playout and anchored jitter.

4.1. Optimum Playout of Frames

The automaton *Optimum Playout* (Fig. 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.

4.2. 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 Fig. 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.

4.3. Non-anchored Jitter

An automaton that generates a stream exhibiting non-anchored jitter is depicted in Fig. 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.

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 Fig. 6 (right). While running, the clock

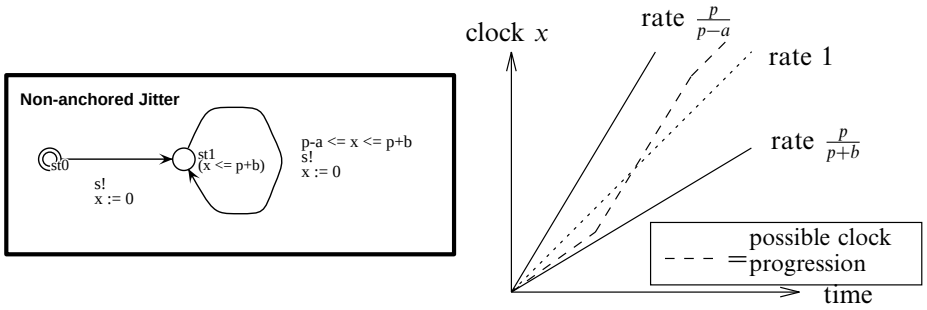


Fig. 6. Timed automaton for non-anchored jitter and fluctuating clock rate.

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 [OSY94]. Indeed, if we apply this transformation on our hybrid automaton we obtain a timed automaton that is equivalent to the automaton *Non-anchored Jitter*.

5. Formal Modelling of Timeout

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

5.1. Bounded Timeout

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 [NiS91].

In the context of this paper it is assumed that if action *good* is enabled before time D (since timeout activation time), then it will occur by time D; actually, we require that action *good* must be executed as soon as it is enabled, i.e. it must be urgent. In the domain of multi-media presentation this assumption is justified by the observation that if a frame is available it should be processed as soon as possible. If the frame is available, it makes no sense to delay such a processing beyond the timeout deadline without reason.

In Fig. 7 (left) it is shown how a bounded 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 transition where the clock x 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

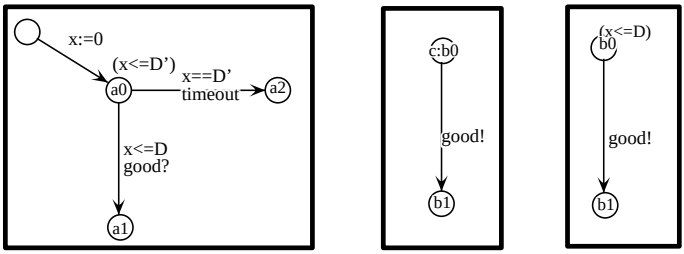


Fig. 7. Modelling a bounded timeout.

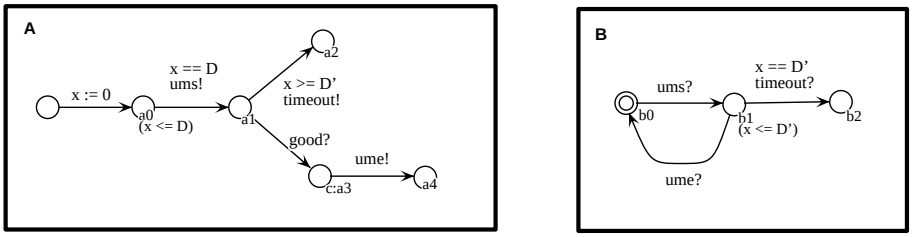


Fig. 8. Modelling a precise timeout.

$x = D'$ at the transition labelled by *timeout*, and the invariant at location $a0$ that requires $x \leq D'$. The fact that *good* should happen at latest when x reaches the value D is expressed by the guard $x \leq D$ at the transition from $a0$ to $a1$.

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. Urgency has to be guaranteed by other means. As pointed out in Section 3, UPPAAL provides three ways for expressing urgency: committed locations, urgent channels and through combinations of guards and invariants. Unfortunately, *good* cannot be defined as an urgent channel because in the timeout construction it is guarded by $x \leq D$ 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 invariants. These committed locations and invariants will belong to automata that synchronise via *good* with (the automaton containing) the timeout; both possibilities are depicted in Fig. 7 (two rightmost automata).

It is important to point out here that such committed locations and invariants must be used with care, since they can easily generate dead/time-locks. In particular, once a *timeout* has occurred, it is no longer possible for a *good* action to occur.

5.2. Precise Timeout

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 8 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 remark 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

location 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 is 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 precise timeout is modelled by the interaction between the automata A and B that share clock x (see Fig. 8). Suppose B is in its initial location. The timeout is activated by moving to location $a0$ in automaton A while resetting clock x. Automaton A stays in location $a0$ until x reaches the value D. When $x = D$, automaton A moves to location $a1$. By synchronisation on *ums*, automaton B moves to location $b1$ at the same time. According to the guard in automaton A, *timeout* is possible at any time at least D'. Automaton B, however, is only able to perform *timeout* (while moving from $b1$ to $b2$) at time D'. The joint execution of *timeout*, if it happens, is thus ensured to happen at $x = D'$.

On the other hand, action *good* is allowed to occur at any time before the *timeout* occurs. When *good* happens, we have to reset automaton B in order to avoid a time-lock. This is established by making location $a3$ committed, which will force the transitions from $a1$ to $a3$, $a3$ to $a4$ and $b1$ to $b0$ to occur in one atomic step.

5.3. Remarks

A few considerations are now due. First of all, we remark that our UPPAAL model of the precise timeout tolerates an occurrence of the *good* action at any time satisfying $D \leq x \leq D'$, whereas we required that such an action should be allowed only when $x = D$. Moreover, when $x = D'$ both *good* and *timeout* may occur, thus we have indeed modelled a *weak* timeout in the terminology of [NiS91]. The first problem has again to do with the fact that one is not allowed to associate a guard (like $x = D$) with a transition labelled by an action on an urgent channel (like *good*). A similar situation arises with invariants on locations that act as source for a transition with an urgent channel. This implies that *good* can happen at any time until the timeout expires, including the case $x = D'$. In any case, if the *good* action is available when $D \leq x < D'$, then it is *guaranteed* to happen, and this is the maximum we can guarantee with this model of precise timeout. Thus, it is not possible to model a precise timeout in UPPAAL [Yi98]. Probably, some notion of priority could help in these situations, but is not provided by UPPAAL.

6. Formal Modelling of the Lip-Synchronisation Protocol

In this section we give a formal specification in UPPAAL of the LSP. This specification is based on the timed LOTOS specification given in [Reg93] which covers the behaviours of the video, the sound managers and the synchroniser (see Fig. 2).

The full UPPAAL specification is shown in Fig. 9, where the *Video Manager*, the *Sound Manager*, the *Video Watchdog* and the *Sound Watchdog* are modelled respectively by the automata *VideoMgr*, *SoundMgr*, *VideoWdg* and *SoundWdg* (together with *UrgMon*). The *Synchroniser* is composed of automata *Sync*, *VideoSync*, *SoundSync* 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 briefly discuss the various components.

6.1. The Stream Managers

The sound and video managers are triggered by the availability (at the presentation device) of a sound or video frame, respectively. This is modelled by the actions *savail* and *vavail*. The availability of a media frame is immediately reported to the synchroniser 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 *sm2* and *vm2* as committed. The managers must then wait for an indication from the controller (actually the watchdogs) that the media frame is to be presented. This is modelled by the actions *sokk* and *vokk*. As soon as the indication has been obtained, the presentation device must be given a signal to present the frame. This is modelled by the internal actions *spresent* and *vpresent*. These actions are left internal because the presentation device itself is not further specified.[5]

6.2. The Watchdog Timers

Each watchdog timer ensures that the time between two subsequent presentations of a frame of the same kind is between certain bounds. If a frame 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 *vok*. This action precedes *vpresent*, but it is guaranteed that no time passes between *vok* and the presentation of the video frame. When the first *vok* is observed, clock $x4$ is started and action *vokk* is issued to the video manager without any delay. The combination of *vok* and *vokk* establishes the synchronisation between three automata, namely the automata *VideoMgr*, *VideoWdg* and *VideoSync*. 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 $x4 >= 35$ and $x4 <= 45$. When *vok* occurs, clock $x4$ is reset. Immediately after *vok* there is a transition, from a committed location, labelled by *vokk* back to location *vw3* to start a new timeout session. If *vok* does not occur before 45 ms have passed, a *vlate* error is issued at time $x4 == 46$. 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 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 — one ms after its original presentation time. The initial part of *SoundWdg* is similar to that of *VideoWdg*. When the first *sok* is observed, timer $x3$ is started and synchronisation with the *SoundMgr* is established via *sokk*.

[5] In [Reg93] there is an additional action *presented* that is performed by the presentation device to mark the end of the presentation of a frame. We have omitted this action, because in the timed LOTOS specification this action was apparently assumed to occur always before the next frame becomes available and therefore cannot create further complications for the protocol.

The repeated timeout construction is a precise timeout, as discussed in Section 5. The *SoundWdg* waits in location *sw3* until 30 ms have passed since the last appearance of a sound frame. At that time it notifies the urgency monitor *UrgMon* by means of action *ums* and it resets clock $x3$. 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 one ms later. The construction with *UrgMon* is needed to guarantee that *slate* happens urgently (i.e. it has the role of automaton *B* in Figure 8). If the *sok* happens in time, *UrgMon* is immediately notified by *ume* and a *sokk* action is generated to model the multi-party synchronisation between the *SoundMgr*, *SoundWdg* and *SoundSync*.

6.3. The Synchroniser

The synchroniser *Sync* is activated by *vready* or *sready*. Depending on which of these actions occurs first it generates a *vok* or a *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. In order to start the automata in the right way their initialisation is synchronised on special actions that do not occur in the timed LOTOS specification of [Reg93]. 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 shorthands of the following:

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

Note that all the locations except the initial location of the *Sync* 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.

6.4. The Sound Clock

The *SoundClock* is a discrete clock that generates one tick per ms. It is started when the first sound frame has arrived and is presented. This clock serves as a reference time to compute the amount of skew between the sound and the video stream. If a sound frame arrives first, the clock is started via *sti*. During this transition a variable *vmins* is updated that keeps track of the amount of skew between the sound and the video stream.[6] If a video frame arrives first, the *SoundClock* is started by means of the *std* action. In this case the clock starts after synchronisation on action *sclock* and indicates the arrival of the first sound frame.

[6] This variable is called *vmins* because of its direct relation to the original timed LOTOS specification [Reg93] 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.

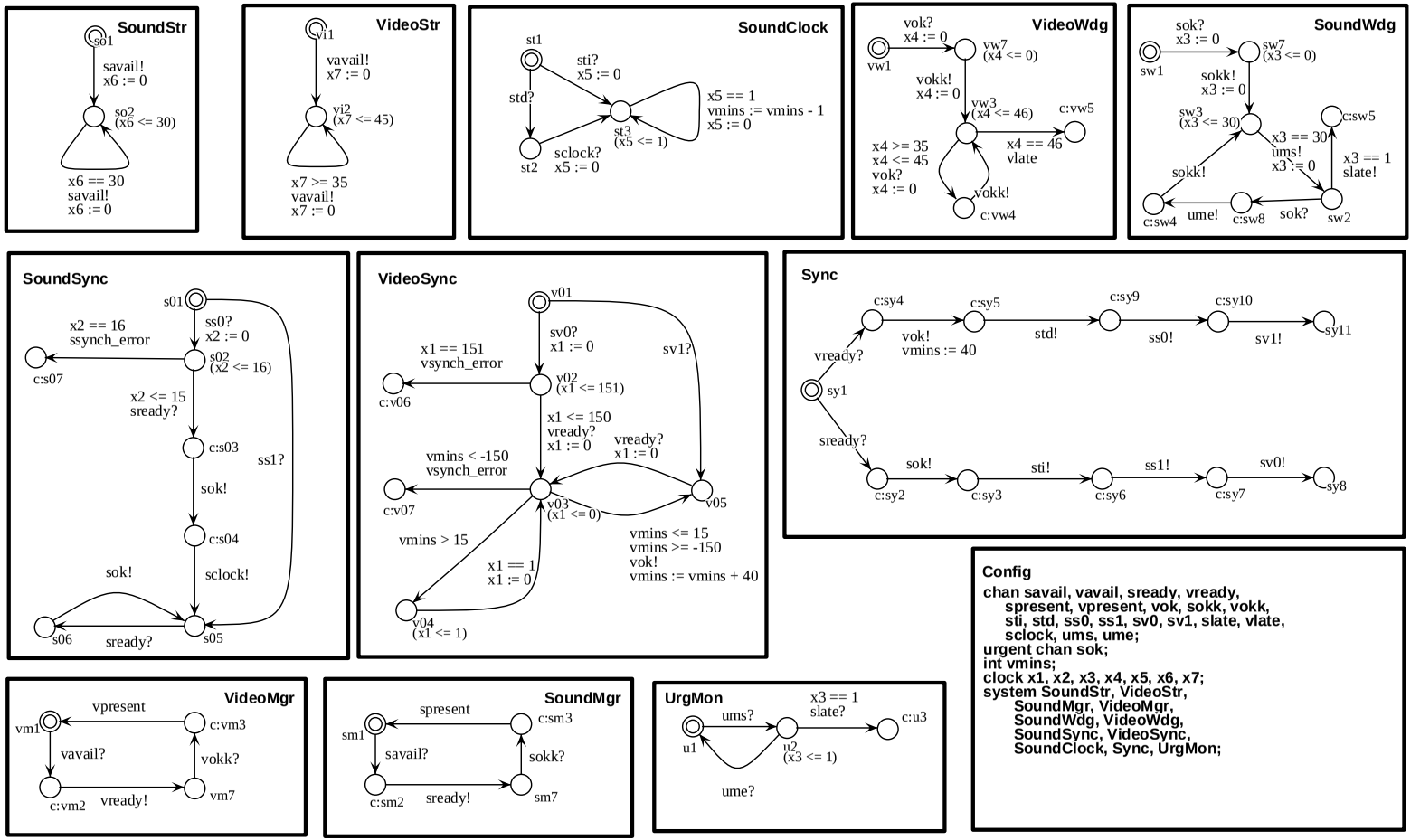


Fig. 9. Modelling the LSP in UPPAAL.

6.5. The Sound Synchroniser

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. 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.

6.6. The Video Synchroniser

The video synchroniser is the most complex process of the LSP. 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 *VideoSync* waits for a *vready* action. When it receives a *vready* it resets clock $x1$ and goes to location *v03* where it checks the lip-synchronisation requirement immediately (due to the invariant $x1 <= 0$). 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 location in which the video synchroniser is forced to wait one ms and then repeats the checking of the lip-synchronisation requirement.
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.

6.7. The Media Streams

Since the informal specification does not describe any assumptions about the streams, we can in principle model them as we like. The models of the media streams we used are further described in the next section.

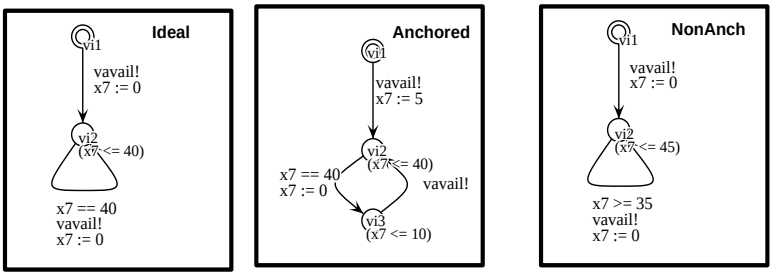


Fig. 10. Three types of video stream behaviour.

7. Verification

7.1. Verified Properties

7.1.1. Assumptions

For any model of the sound stream that does not let frames arrive every 30 ms the LSP does not behave properly. It seems clear that the LSP 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 exhibit any form of jitter.

Instead of verifying properties that have been reported in the literature on lip-synchronisation, such as proving that a sound frame is presented every 30 ms [FBS96], we take such basic properties for granted and we explore possible problems caused by jitter of the video stream.

7.1.2. Types of Video Streams

In the specifications of the LSP found in the literature [SHH92, Reg93, BBB94, BBB97, FBS96] 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. We investigated the protocol for three types of video stream behaviour:

- An ideal (optimum playout) 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 streams are modelled by the timed automata depicted in Fig. 10. These automata are just instantiations of the automata for modelling jitter as shown in Section 4. For each video stream behaviour two scenarios have been investigated: a scenario in which the start of each stream was left unspecified (possible initial relative delay) and a scenario in which both streams start at the same time (no initial relative delay).

7.1.3. Reachability of Error Locations

In the verification we did a reachability analysis of the error locations. The model of the LSP contains the following error locations:

- Initial sound synchronisation error in the *SoundSync* (location $s07$)
- Initial video synchronisation error in the *VideoSync* (location $v06$)
- Video synchronisation error in the *VideoSync* (location $v07$)
- Video late error in the *VideoWdg* (location $vw5$)
- Sound late error in the *SoundWdg* (location $sw5$ and $u3$)

The results for each situation have been obtained by checking the reachability property on the model consisting of the LSP and a variant of the video stream. The reachability properties we checked are all of the form

$$E <> (A.l \text{ and not } (B_1.l_1 \text{ or } \ldots \text{ or } B_n.l_n))$$

i.e. does there exist a path in which eventually automaton A is in location l and the other automata B_i are not in their locations l_i. Here, location l is the location that indicates the error the reachability of which we are checking. The second conjunct ensures that the other automata (B_i) did not reach another error location. In this way, it is ensured that the verified error did not occur as a consequence of other errors. It is worth recalling here that all error locations of our automata are sink locations.

7.2. Verification Results

7.2.1. System Configuration

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 experienced 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. This forces a time-lock whenever any such location is reached.

7.2.2. Scenario with Possible Initial Relative Delay

Table 1 gives the result of verifying the LSP for the various reachability properties in case 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 (T or F) and the CPU time in seconds are reported. The numbers between brackets after each T indicate the number of time units (ms) that are needed to reach the error.

From the first two rows of the table it is clear that initial out of synchronisation errors for both the video and the sound can always occur. This is due to the fact that the time between the first video and the first sound frame can be arbitrarily long. The error occurs exactly when the maximal delay has elapsed, i.e. at 16 ms

Table 1. Verification results for scenario with initial relative delay between streams.

Property	*Ideal video*		*Anchored video*		*Non-anchored video*	
	Result	Time	Result	Time	Result	Time
Init Sound Sync err	T (16)	0.06	T (16)	0.08	T (16)	0.06
Init Video Sync err	T (151)	147.09	T (151)	6336.79	T (151)	225.37
Video Sync err	F	293.71	T (191)	16004.86	T (191)	370.71
Video Late	F	293.66	T (81)	473.61	F	2095.87
Sound Late	F	294.25	F	32741.62	F	2640.95
Deadlocks	1		1		1	

Table 2. Verification results for scenario without initial relative delay between streams.

Property	*Ideal video*		*Anchored video*		*Non-anchored video*	
	Result	Time	Result	Time	Result	Time
Init Sound Sync err	F	1.08	F	23.12	F	2649.62
Init Video Sync err	F	1.08	F	23.16	F	2652.44
Video Sync err	F	1.07	F	23.19	T (1391)	2618.70
Video Late	F	1.08	T (81)	3.96	F	2648.42
Sound Late	F	1.08	F	23.14	F	2661.94
Deadlocks	None		None		None	

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 UPPAAL verifier `verifyta` performs a complete state space search in about 293 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. Its 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.

7.2.3. Scenario without Initial Relative Delay

When both streams are forced to start at the same time the results of the verification are rather different, as shown in Table 2. 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 UPPAAL verifier `verifyta` reported any deadlocks which were not caused by reaching an error location. When the video and the sound start at the same time no deadlocks were reported. When they start independently one deadlock 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 was modelled in *VideoSync* 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 $x1$ is beyond 150 ms in location $v02$ of *VideoSync*. This leads to a time-lock. This kind of time-lock has already been highlighted in Section 5. In the original LSP described in [Reg93] this problem could not occur because a discrete time model was used. This time model implicitly 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 *SoundSync* but even a full state space search did not reveal it. We think that further research is required to find the cause of this.

With the anchored video the verifier `verifyta` did not report any further deadlock either, but by means of the simulator `simta` we have found a time-lock just after a few transitions from the starting location. This time-lock 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 *VideoSync* needs to synchronise 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 $x7 < 45$ leads to a time-lock situation. This time-lock is a very interesting one because it reveals another, quite hidden, problem of the LSP.

The time-lock occurs when *VideoStr* has to synchronise on *vavail* with *VideoMgr* just before 45 ms passed, and *VideoSync* is at location $v04$ because the video was too early with respect to sound (i.e. $vmins > 15$). In this situation the synchronisation on *vavail* cannot take place, because *VideoMgr* is at location $vm7$ waiting to synchronise 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 $(x1 == 1)$ on the outgoing transition at location $v04$ in *VideoSync*. Due to this forced delay, the invariant $x7 < 45$ at location $vi2$ of *VideoStr* cannot be satisfied, thus leading to the time-lock. In the non-strict version $(x7 \leq 45)$ this time-lock is avoided in a curious way. The synchronisation on *vavail* between *VideoStr* and *VideoMgr* is *delayed* until $x7 == 45$ so that *VideoSync* can leave location $v04$ by pure time passing and subsequently synchronise on *vok* with *VideoWdg*. This enables the synchronisation on *vokk* between *VideoWdg* and

VideoMgr and *vpresent* to occur at the *VideoMgr*. Since the complete sequence of transitions, after *VideoSync* has left location $v04$, occurs without consuming time because of the concatenation of committed locations, the synchronisation on *vavail* between *VideoStr* and *VideoMgr* can take place as well.

This aspect of the LSP 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 LSP 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-synchronisation protocol (LSP) using UPPAAL. Specifications of this protocol have been presented in a number of different formalisms [SHH92, BBB94, BBB97, Reg93, FBS96]. We have particularly followed the timed LOTOS specification of [Reg93].

8.1. Specifying and Verifying the LSP

We found it interesting to investigate how several typical multi-media concepts, like jitter, drift and skew can be formally specified and analysed using timed automata. Our verification has identified a number of interesting issues with the protocol, of which, two of the most important are:

- The last column of Table 2 indicates that with non-anchored jitter, which was the type of jitter the protocol was apparently designed for, and both streams starting together, lip synchronisation cannot be guaranteed for more than *just over one second* (1391 ms). This is clearly quite a low figure. However of course, if we reduced the amount of perturbation 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 synchronisation for a short period of time, our verification work has also highlighted some concrete problems in the protocol. In particular, we have shown that with all types of video streams we defined, a time-lock can be reached in which none of the components is in a prescribed error state. Some of these time-locks have been found automatically, others were found by simulation. Some time-locks appeared because we used a dense time model to describe a protocol previously specified in a discrete time model. Other time-locks 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 LSP is that it has very limited capabilities for keeping the media streams synchronised. This limitation could partially be overcome by means of buffering, i.e. the possibility that the presentation device smoothes out

synchronisation errors by buffering frames before playing them. We are currently investigating the possibility to add such buffering.

8.2. Experience with UPPAAL

The work reported here has also enabled us to evaluate the UPPAAL tool in the context of a non-trivial specification and verification scenario in the area of media synchronisation. Our experience with UPPAAL has generally been positive. Nonetheless we can point out 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 [BFM98]. However, the strategy is not implemented yet and thus, has to be performed by hand.
- *Time-locks.* 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 time-lock can arise without being in a prescribed error location. However, we have not identified all these states through direct verification for time-lock freedom. In particular, one of them was not revealed by the model checker, but rather through simulation. Why some time-locks have not been reported during full state space search of the specification is not clear to us and requires further research. It would be interesting to repeat our experiment with some other tool like KRONOS [DOT96]. KRONOS accepts a richer set of temporal logic formulae, including "unbounded" liveness properties. Freedom from time-locks 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 rather effective in the context of the LSP, which contains a considerable number of clocks.
- *Low-level notation.* Timed automata can 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 time-locks. It would be interesting to have a set of generic high-level operators for timed specification, which could be mapped onto timed automata, e.g. following the ideas proposed in [DaB96] or using hierarchical automata. Furthermore, the timed automata notation only allows parallel composition at "top level", i.e. component automata cannot themselves contain parallel compositions. This leaves a question mark over the scalability of the notation.
- *Stochastic time.* In timed automata, time values are either fixed deterministically or completely non-deterministic. For several purposes it is of interest to quantify the probability of presentation at a certain time instant. *Stochastic automata* [DKB98] allow for the description of time values as random variables with general distributions. Samples of such variables are assigned to clocks which run backwards. Clock expirations (i.e. a clock has reached value

0) can be used as guards. Invariants are absent: transitions are taken as soon as they are enabled. We think that stochastic automata could be very useful for describing many concepts of multi-media systems like, for instance, jitter and skew. It would be interesting to investigate model checking algorithms for these type of automata and integrate these into UPPAAL.

Acknowledgements

We would like to thank Wang Yi and Paul Pettersson (both of Uppsala University) for advice on UPPAAL and Stavros Tripakis (VERIMAG-SPECTRE) for fruitful discussions on symbolic model checking. In addition, David Duke (University of York) was involved in some preliminary work on verifying the LSP. The first author is currently on leave at CNR-Istituto CNUCE under the support of the ERCIM Fellowship Programme. The last author is supported by the TACIT network under the European Union TMR Programme, contract ERB FMRX CT97 0133.

References

[AlD94] Alur, R. and Dill, D.: A theory of timed automata. *Th. Comp. Sci.*, **126**, 183–235 (1994).

[BGK96] Bengtsson, J., Griffioen, W. O., Kristoffersen, K. J., Larsen, K. G., Larsson, F., Pettersson, P. and Wang Yi: Verification of an audio protocol with bus collision using UPPAAL. In Alur, R. and Henzinger, T. A., eds, *Computer-Aided Verification VIII*, LNCS 1102, pp. 244–256. Springer-Verlag, 1996.

[BBB97] Blair, G. S., Blair, L., Bowman, H., Chetwynd, A.: *Formal Specification of Distributed Multimedia Systems.* University College London Press, 1997.

[BBB94] Bowman, H., Blair, G. S., Blair, L., Chetwynd, A.: A formal description technique supporting expression of quality of service and media synchronisation. In Hutchison, D., Danthine, A., Leopold, H., and Coulson, G., eds, *Multimedia Transport and Teleservices*, LNCS 882, pp. 145–167. Springer-Verlag, 1994.

[BFM98] Bowman, H., Faconti, G., and Massink, M.: Specification and verification of media constraints using UPPAAL. In Markopoulos, P. and Johnson, P., eds, *Design, Specification and Verification of Interactive Systems.* Computer Science Series, pp. 261–277. Springer-Verlag, 1998.

[DaB96] D'Argenio, P. R. and Brinksma, E.: A calculus for timed automata. In Jonsson, B. and Parrow, J., eds, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 110–130. Springer-Verlag, 1996.

[DKB98] D'Argenio, P. R., Katoen, J.-P., and Brinksma, E.: An algebraic approach to the specification of stochastic systems (extended abstract). In Gries, D. and de Roever, W.-P., eds, *Programming Concepts and Methods*, pp. 126–148. Chapman & Hall, 1998.

[DKR97] D'Argenio, P. R., Katoen, J.-P., Ruys, T. C., and Tretmans, G. J.: The bounded retransmission protocol must be on time! In Brinksma, E., ed, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1217, pp. 416–431. Springer-Verlag, 1997.

[DOT96] Daws, C., Olivero, A., Tripakis, S., and Yovine, S.: The tool KRONOS. In Alur, R., Henzinger, T. A., and Sontag, E. D., eds, *Hybrid Systems III*, LNCS 1066, pp. 208–219. Springer-Verlag, 1996.

[FBS96] Feyzi Ates, A., Bilgic, M., Saito, S., and Sarikaya, B.: Using timed CSP for specification, verification and simulation of multimedia synchronization. *IEEE J. on Sel. Areas in Comm.*, **14**, 126–137 (1996).

[FiL98] Fischer, S., and Leue, S.: Formal methods for broadband and multimedia systems. *Comp. Netw. and ISDN Sys.*, **30**(9/10), 901–924 (1998).

[JLS96] Jensen, H. E., Larsen, K. G., and Skou, A.: Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. *Proc. of the 2nd SPIN Workshop*, Rutgers DIMACS Series, pp. 1–20, 1996.

[LPY97] Larsen, K. G., Pettersson, P., and Wang Yi: UPPAAL in a nutshell. *Int. J. on Software Tools for Technology Transfer*, **1**(1/2), 134–152 (1997).

[Lin95] Linington, P. F.: RM-ODP: The architecture. In Raymond, K., Armstrong, L., eds, *Proc. IFIP TC6 Int. Conf. on Open Distributed Processing*, pp. 15–33. Chapman & Hall, 1995.

[Mil89] Milner, R.: *Communication and Concurrency*. Prentice-Hall, 1989.

[NiS91] Nicollin, X., and Sifakis, J.: An overview and synthesis on timed process algebras. In de Bakker, J. W., Huizing, C., de Roever, W.-P., and Rozenberg, G., eds, *Real-Time: Theory and Practice*, LNCS 600, pp. 526–548. Springer-Verlag, 1991.

[OSY94] Olivero, A., Sifakis, J., and Yovine, S.: Using abstraction techniques for the verification of linear hybrid systems. In Dill, D. L., ed, *Computer Aided Verification VI*, LNCS 818, pp. 81–94. Springer-Verlag, 1994.

[Reg93] Regan, T.: Multimedia in temporal LOTOS: A lip synchronisation algorithm. In Danthine, A., and Leduc, G., eds, *Protocol Specification, Testing and Verification XIII*. North-Holland, 1993.

[SHH92] Stefani, J.-B., Hazard, L., and Horn, F.: Computational model for distributed multimedia applications based on a synchronous programming language. *Comp. Comm.*, **15**(2), 114–128 (1992).

[Ste96] Steinmetz, R.: Human perception of jitter and media synchronization. *IEEE J. on Sel. Areas in Comm.*, **14**(1), 61–72 (1996).

[YPD94] Wang Yi, Pettersson, P., and Daniels, M.: Automatic verification of real-time communicating systems by constraint solving. In Hogrefe, D., and Leue, S., eds, *Formal Description Techniques VII*. Chapman & Hall, 1994.

[Yi98] Wang Yi: Personal communication. 1998.

Received March 1998
Accepted in revised form October 1998