Specification and Verification of Media Constraints using UPPAAL*

Howard Bowman[1], Giorgio P. Faconti[2] and Mieke Massink[3]

[1] Computing Lab., U. of Kent, Canterbury, Kent, CT2 7NF, UK
[2] CNR-Istituto CNUCE, Via S.Maria 36, 56126 - Pisa - Italy
[3] Dept. of Computer Science, U. of York, Heslington, York, YO1 5DD, UK

H.Bowman@ukc.ac.uk, G.Faconti@cnuce.cnr.it and M.Massink@guest.cnuce.cnr.it

Abstract. We present the formal specification and verification of a multimedia stream. The stream is described in a timed automata notation. We verify that the stream satisfies certain quality of service properties, in particular, throughput and end-to-end latency. The verification tool used is the real-time model checker UPPAAL.

1 Introduction

The acceptance and utility of a broad range of application systems is substantially affected by their ability to present information in an effective and appealing way to human users. Rapid progress in the development of multimedia technology promises more efficient forms of man/machine communication. However, the use of multimedia for conveying information does not guarantee effective and intelligible presentations per se. Appropriate design decisions must be drawn that lead to a fine grain coordination of communication media and modalities. This may even become a harder and more complex task than solving the application problem. Furthermore, in the vast majority of non-trivial applications the information needs will vary from user to user and from situation to situation. Consequently, a multimedia system should be able to flexibly generate various presentations for one and the same information content in order to meet individual requirements of users and situations, resource limitations of the computing system, and so forth.

In this respect, technology based techniques and models are just one out of many components contributing to the design of an interactive system. Other disciplines provide complementary views and perspectives on design problems and those contributions need to be related to one another to help support design reasoning and decisions. Different kinds of analysis contribute to augment the comprehension of a particular design space either by contributing complementary information or by providing criteria addressing equivalent requirements. In addition, some design issues may be raised by one particular kind of analysis

* The first author is currently on leave at CNUCE under the support of the European Research Consortium for Informatics and Mathematics (ERCIM).

from a discipline, but only solved if another kind of analysis is applied from another discipline.

In such a scenario, many system requirements are generated and derived from disciplines distant from technology and computer science such as semiotics and cognitive psychology. With the current state of the art in system design, these requirements cannot be bounded in a straightforward manner to a specific technology but require that system designers be informed both of *why* requirements have been formulated and *how* they can possibly be satisfied. Techniques for representing design decisions, and frameworks for communicating and contextualizing more analytic approaches into the practicalities of design exist already in the literature of both software engineering and human/computer interaction. Although significant conceptual progress has been made in both of these directions, their general application requires us to understand how the basic approaches can actually be used in practical settings; this remains an open issue.

Here, we refer to a particular class of interactive systems, namely Multi-Media Presentation Systems. This class of systems has been characterized in [5] where a Reference Model for Intelligent Multi-Media Presentation Systems is presented, integrating system concerns with the necessary representation of the contextualized knowledge about domain problem, design process, and user modeling. The model is abstract since it identifies the basic components of a presentation system and relates them. The details are addressed by specific models that are located in the abstract model, such as the Presentation Environment for Multimedia Objects [11]. Examples of applications that can be described are teleconferences and automatic generation of multimedia help systems, their distinguishing features being the *continuity* of the involved media, and the kind of *indirect interaction* with the media objects.

Interaction in these systems is indirect since it doesn't concern the manipulation of the media objects themselves (i.e. the video content); rather, it addresses the mechanisms influencing their presentation (i.e. compression/decompression algorithms, allocation of band) and the global quality of service. In [2], for example, it is suggested that audio/video quality of service be controlled by separate sliders each dealing with a specific parameter. The issue is that some parameters reflect perceivable properties of the media objects with no direct link to the underlying technology and it is difficult to predict the demand of computational resources required to achieve the quality of service set by the end-user.

Continuity refers to media streams conveying digital objects with associated time constraints, distinguishing these applications from traditional protocols. Usual techniques such as retransmission of data in case of faults or losses are no longer applicable and the verification of system throughput and latency against the time constraints becomes a must. As an example, Video/Audio streams must satisfy $inter-media\ synchronization$ constraints as in the case of the lip-synchronization. However, continuity plays also an important role in the context of a single media stream since it demands $intra-stream\ synchronization$, i.e. the relationship between media objects within the same data stream thus adding to throughput and latency a further constraint on jitter.

Consequently, timing plays an important role in multimedia presentation systems. The quality and the usability of those systems is crucially dependent on their performance with respect to timeliness. Often there is a trade-off between the quality of the presentation of multimodal information and the quality of service that can be offered by the medium over which the information is transported. Specification and verification of real-time aspects of (multi)media systems is however a difficult problem. The most common approach to building those systems is to implement new ideas directly in a prototype and to measure the performance. This approach is not always satisfactory. The complexity of the behaviour of the distributed algorithms is often considerable and their correctness is difficult to estimate by mere testing. Furthermore, it is often very hard to find the causes of rare errors and to improve the implementation accordingly. This has been illustrated for example in [10] where an Audio/Video protocol has been analyzed that had been developed in an industrial setting.

In this paper, we restrict our analysis to the investigation of a number of issues for the single data stream case. This allows us to set up a basic framework for further work on multi-(inter-)media synchronization while enabling the investigation of the capabilities and the practical use of formal notations and their associated tools with small size experiments.

2 A Simple Media Stream

The most basic requirement for supporting multimedia is to be able to define continuous flows of data; such structures are typically called *media streams* [4]. In this paper, we will discuss and illustrate a number of aspects related to media constraint modeling using such a multimedia stream.

The basic media stream is as depicted in figure 1. It has three top level components: a *Source*, a *Sink* and a *communication Medium* (which we will from now on simply refer to as the *Medium*). The scenario is that the *Source* process generates a continuous sequence of packets[1] which are relayed by the *Medium* to a *Sink* process which displays the packets. Three basic inter-process communication actions support the flow of data (see figure 1 again), *sourceout*, *sinkin* and *play*, which respectively transfer packets from the *Source* to the *Medium*, from the *Medium* to the *Sink* and display them at the *Sink*.

Formal descriptions of media streams have been given before, e.g. [4] etc. However to our knowledge, no formal verifications have been performed. This is one contribution of this paper.

The following informal description of the behaviour of the stream is kept similar to the LOTOS/QTL specification that appears in [4].

- All communication between the *Source* and the *Sink* is asynchronous.
- The *Medium* is unreliable; it may loose and reorder packets.

[1] These could be video frames, sound samples or any other item in a continuous media transmission. In this way the scenario remains completely generic. However, instantiation of data parameters will specialize the scenario.

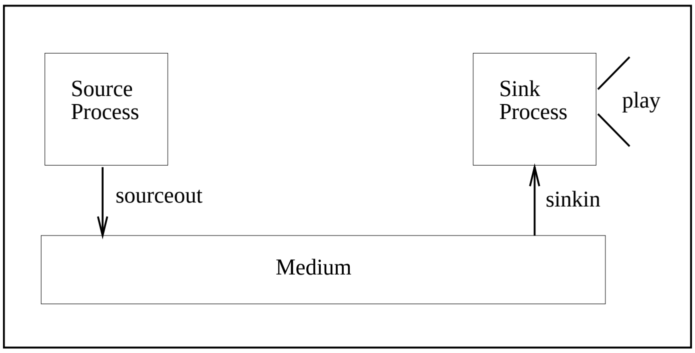


Fig. 1. A Multimedia Stream

- The *Source* transmits a packet every 50 ms (i.e. 20 packets per second).
- Packets that do not get lost arrive at the *Sink* between 80 ms and 90 ms after their transmission. This is the latency of the *Medium*.
- Whenever the *Sink* receives a packet, it needs 5 ms to process it, after which it is ready to receive the next packet.

In section 4, we will present an UPPAAL description of this basic behaviour. Then we focus on our main objective: to check that this system satisfies certain quality of service properties. Conceptually, these properties can be viewed as being derived from user presentation requirements. The quality of service properties that we wish to verify are:

1. **Throughput.** We would like to ensure that the *Sink* process receives packets at the rate of between:

   > 15 and 20 packets per second.

   Clearly, there is a direct link between the rate of loss of the *Medium* and the throughput at the *Sink*. Thus, the flavour of our investigation of this property will be to determine what are the bounds on the rate at which the *Medium* looses messages in order to satisfy this throughput property.
   We will also build into the system the possibility that it can go into an error state and halt, if the throughput property is invalidated.
2. **Latency.** We will check the following latency property:

   > The end-to-end delay between a *sourceout* action and its corresponding *sinkin* action cannot be more than 95ms.

   which puts an upper bound on the end-to-end transmission delay.
3. **Jitter.** Jitter is defined as the variance of delay. It ensures that there is not an unacceptable variability around the optimum presentation time, e.g. if one packet is presented quite early and the next is presented relatively late an unacceptable stutter in the presentation will result. A full analysis of jitter would require stochastic techniques [9] to be employed. This is clearly not

possible with the verification technology we have available to us. However, placing both an upper and lower bound on the latency would impose a crude bound on jitter. Apart from noting that we could extend our latency analysis to do this, we do not consider the property any further in this paper.

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 modeled as a network of extended timed automata with global real-valued clocks and integer variables. The behaviour of a network of automata can be analyzed by means of the simulator and reachability properties can be checked by means of the model checker.

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. It 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 analyzed 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 labeled. 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* on clocks and data variables expressing under which condition the transition can be performed. Absence of a guard is interpreted as the condition *true*.
- a synchronization or internal *action* that is performed when the transition is taken. In case the action is a synchronization action then synchronization with a complementary action in another automaton is enforced following similar synchronization rules as in CCS [14]. Given channel name **a**, **a!** and **a?** denote complementary actions corresponding to *sending* respectively *receiving* on the channel **a**. Absence of a synchronization action is interpreted as an internal action similar to τ-actions in CCS.
- a number of *clock resets* and *assignments to integer variables*

The label of locations consists also of three parts:

- the *name* of the location which is obligatory.
- an *invariant* expressing constraints on clock values, indicating the period during which control can remain in that particular location.
- an optional marking of the location by putting **c:** in front of its name indicating the location as *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 or interleaving of other actions.

In the configuration, the following aspects of the network are defined:

- declarations of global clock and integer variables
- the channel names that are the names of the actions. Channels can be defined as normal communication channels or *urgent* channels. When a channel is urgent no timing constraints can be defined on the transition labeled 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.
- the list of names of automata the system is composed of.

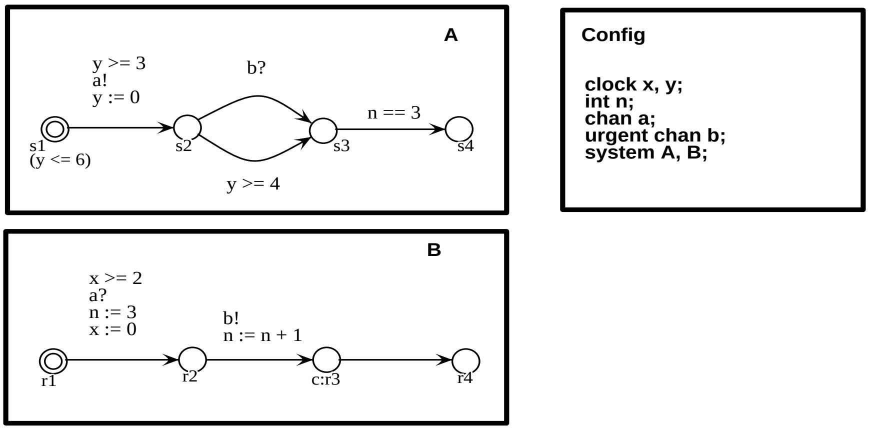
Formally, the states of an UPPAAL model are of the form $(\bar{l}, v)$, where $\bar{l}$ is a *control vector* and v a *value assignment.* The control vector indicates the current control location for each component of the network. The value assignment gives the current value for each clock and integer variable. All clocks proceed at the same speed. There are three types of transitions in an UPPAAL model:

Internal transitions Such transitions 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.

Synchronization A synchronization 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.

Delay 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.

An example of an UPPAAL specification is given in Figure 2. 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 synchronized 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.



```
clock x, y;
int n;
chan a;
urgent chan b;
system A, B;
```

Fig. 2. Example of an UPPAAL specification

When control is in *s2* and *r2* the only transition that is possible is the synchronization 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 labeling 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 determined 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 characterized by simple constraint systems which are conjunctions of atomic clock and data constraints. Details on the calculation of these constraint sets for simulation and model checking can be found in [15].

The properties that can be analyzed 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 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 modeling 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. [3, 7, 12].

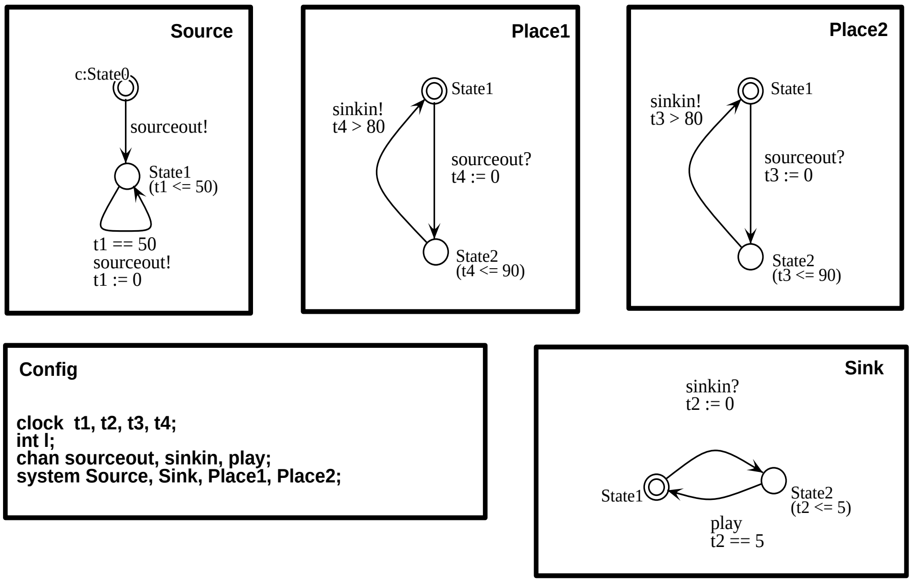
4 Stream example formalized in UPPAAL

4.1 The basic model

Consider the stream example introduced in section 2. The simplest part of the example is the *Source*. In the informal specification it is said that a packet is sent every 50 ms. To make this more precise we assume that the first packet is sent at time equal 0 and all later packets exactly 50 ms one after the other. This behaviour is modeled as an UPPAAL automaton with two locations. The initial location (indicated by a double circle) is annotated as committed to enforce that the first packet (*sourceout!*) is sent immediately. To assure that every following packet is sent exactly 50 ms after the previous one, a clock $t1$ is introduced. The guard $t1 == 50$ enables the sending of *sourceout* at exactly 50 ms after the last one. The invariant at *State*1 enforces that the enabled transition really happens at $t1 == 50$. When the transition is performed $t1$ is reset and the behaviour repeats itself.

The *Sink* is required to always accept a packet, except when it is playing a packet. Whenever it receives a packet it plays it immediately during the next 5 ms. This behaviour is modeled by another automaton with two locations. In the initial location the automaton waits for a packet from the medium. When it arrives a timer $t2$ is set that is used to model the 5 ms delay caused by playing of the packet before control returns to the initial location.

The third part to model is the medium. What is known of the medium is that it acts as an infinite buffer, it has a latency of between 80 ms and 90 ms, it may loose and reorder packets. At first sight we should model the medium as an infinite structure. However, if we are only interested in the throughput of the medium, the order in which packets arrive is irrelevant. What *is* relevant is that we model the medium in such a way that it always allows the *Source* to perform the next *sourceout*. We will show that the medium can be modeled by two independent one-place-buffers. We first model the medium assuming that it does not loose packets. Each buffer is modeled as an automaton with two locations (see Figure 3). At the initial location the buffer can receive a *sourceout* from the *Source*. At that point a timer is started to model the latency of the medium. The *sinkin* action following the *sourceout* is delayed by at least 80 ms and at most 90 ms.



```
clock  t1, t2, t3, t4;
int I;
chan sourceout, sinkin, play;
system Source, Sink, Place1, Place2;
```

Fig. 3. UPPAAL specification of media stream

To be sure that it was correct to model the medium by only two one-place-buffers, we should prove that it is never the case that when the *Source* wishes to perform a *sourceout*, i.e. when $t1 == 50$, both one-place-buffers are full, i.e. they are in location *State*2. In UPPAAL this situation can be formalized as:

$$E <> (t1 == 50 \text{ and } Place1.State2 \text{ and } Place2.State2)$$

which using the UPPAAL model checker can be shown not to hold.

It is important to note however that the minimal number of *Place*s needed to guarantee that the *Source* can always send its packet depends on the time constraints used in the model. If the time between packets at the *Source* would be less than 45 ms there are going to be problems. In our model we can easily verify this by means of the reachability property we formulated before. In this example it is not difficult to find a general formula that gives the minimal number of *Places* needed as a function of the time between the packets and the maximal latency. Let p be the number of *Places*, m the maximal latency and d the interval time between packets sent by the *Source* then:

$$p = \lceil m/d \rceil \qquad (+)$$

In general however, time dependent behaviour can be very hard to predict and a model checker can be helpful to get a good intuition about the relations that hold between parameters of the system. An interesting illustration of this use of model checking can be found in the description of a case study on a bounded retransmission protocol [7].

4.2 Modeling a medium with losses

In this section we relax the assumption that the medium does not loose packets. We assume that the losses are limited to not more than 4 packets per second.

To model this we need a *Monitor* that keeps track of the passing seconds and we need to adapt the automata modeling the *Places*. We model the loss of a packet by an additional internal transition from *State*2 to *State*1 in the automata that model the *Places*. Further we add a global variable l that records the number of losses. The transition is guarded by a constraint on the maximal number of losses. Figure 4 shows the new medium (in fact, this specification contains more additions than just the new medium; these will be explained in the next section).

5 Verifying Quality of Service with UPPAAL

In this section we investigate how the quality of service properties identified earlier in this paper can be verified using UPPAAL.

5.1 Throughput

Let us assume that the throughput of the medium is checked every second and when it is below the threshold of 15 packets per second an *error* is signaled.

The number of packets that arrive at the *Sink* is counted by a global variable x which is updated every time a *sinkin* action occurs. The *Monitor* checks the variable x every second. If the throughput is sufficient then x is reset and the timer starts again. If the throughput is too low an urgent action *error* is generated. As soon as the *Sink* reaches state *State*1 it will synchronize on the *error* action and the media stream will be stopped. In Figure 4 the automata for the *Monitor* and the *Sink* are shown.

The *Monitor* that we present only forces an *error* if too few frames arrive, i.e. if $x < 15$, this is because the possibility of too many frames arriving, i.e.

$x > 20$, cannot arise because of the parameters of the system. However, if it was necessary we could easily add an extra branch in the transition system which caters for this situation.

In addition, we can use UPPAAL to determine the parameters that bound our throughput property. Specifically, we can check under what circumstances our specification satisfies the formula:

$$E <> (Sink.Stop)$$

Satisfaction of this formula implies that our throughput requirement does not hold. As suggested earlier, the obvious parameter that affects throughput is the rate of loss in the medium. Using UPPAAL we can show that if the constraint $l < 4$ is associated with the *loss* action in the medium, as it is in figure 4, then the *Stop* state can be reached, and UPPAAL provides a sample trace. However, if we change the constraint to $l < 3$ then the *Stop* state cannot be reached. Thus, this gives us a clear bound on the number of errors that an acceptable medium should allow.

A subtle point that arises from this specification is that since *State*1 in *Sink* has two outgoing transitions, *error* and *sinkin*, if *State*1 is reached at a time point in which both transitions are enabled then even though *error* is an urgent action, either transition may be taken. This would clearly be undesirable as one would like the system to stop as soon as it is in *error*. However, using UPPAAL an analysis can be made that shows that as long as the *Source* is transmitting at a rate of a frame every 50 ms then this situation cannot arise (interestingly, if it was transmitting at the rate of a frame every 53 ms the situation could indeed arise).

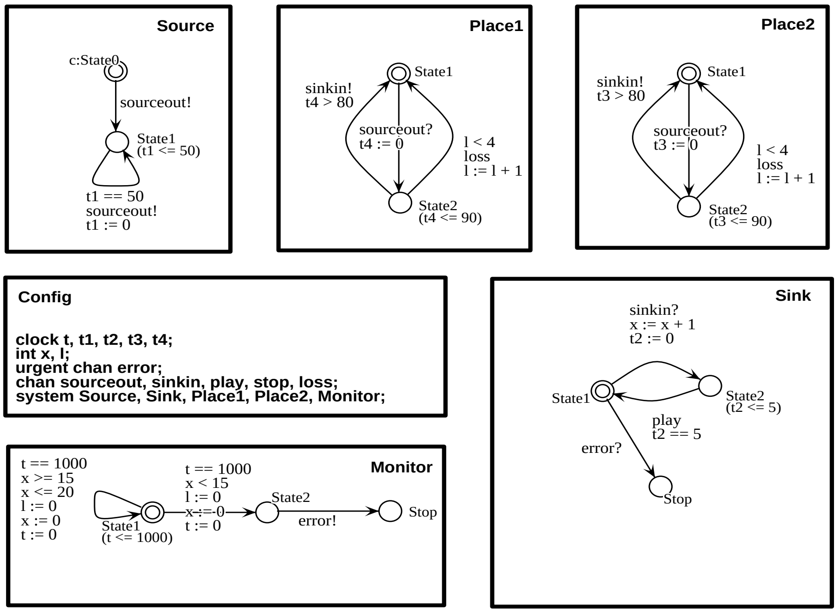
5.2 Verifying Latency

Although in fact, upper and lower bounds for latency of the stream can be very easily discerned by inspection from the automata specification given, this will not always be the case. In fact, in real world systems, communication mediums have highly complex real-time behaviour. For example, there may be a number of different potential routes that frames can take, each accumulating very different latency delays. Furthermore, in the presence of congestion, analysis of latency is far from straightforward. Thus, even though analysing latency is rather superfluous in our stream scenario, it is a valuable exercise to determine the suitability of UPPAAL in this respect.

The first thing to note is that in order to express our latency requirement we must relate corresponding *sourceout* and *play* actions. In order to do this some means of identifying corresponding packets, e.g. by means of time stamps or sequence numbers, must be included.

So, let us formulate our basic latency property with the required sequence numbering. The obvious property that we would like to verify is:

$$\forall x \in \mathbb{N} \,.\, (\Box(play(x) \Rightarrow \overleftarrow{\Diamond}_{\leq 95}\, sourceout(x)))$$



```
clock t, t1, t2, t3, t4;
int x, l;
urgent chan error;
chan sourceout, sinkin, play, stop, loss;
system Source, Sink, Place1, Place2, Monitor;
```

Fig. 4. UPPAAL specification of mediastream with QoS *Monitor*

where the operators used are linear time temporal logic operators [13], which contrast with the branching time operators used in UPPAAL. $\Box P$ is the always/henceforth operator (i.e. in the future it is always the case that P holds), $\Rightarrow$ is logical implication and $\diamondminus_{\leq t} P$ is a past tense operator stating that, P must hold no more than t time units before the current moment. So, the formula states that it is always the case that, if a *play* occurs then at some point not more than $95ms$ before a *sourceout* must have taken place. The significance of the past operator is that it allows for loss in the system, i.e. it only enforces a timing constraint on the *play*s that arise from successfully received packets.

However, this property cannot be verified using UPPAAL. There are two reasons for this.

1. Infinite data sets, such as the natural numbers; and
2. expressing data passing actions

are not supported at present by UPPAAL.

We will show how to handle the second of these difficulties shortly, but for the moment, let us concentrate on the first. The problem is that we need to bound the set of sequence numbers used and in fact, it will greatly simplify the resulting automaton and the state space explosion problem if we can keep the size of this bound very small.

What we would like to ensure is that we have a sufficient number of sequence numbers that we do not get two packets with the same sequence number in the system at the same time. This is a similar requirement to the bounding of the size of the medium investigated in section 4.1. In fact, we can use formula (+) stated there to derive that two sequence numbers are sufficient in a correctly behaving system. Importantly though, we replace m in the formula with the *desired* latency value rather than the known one.

Having decided that two sequence numbers, i.e. 1 and 0, will be sufficient we have to adapt our automata specification accordingly. Now as already noted, actions in UPPAAL are not data passing however, we can get the effect of data passing actions by including a more discriminating set of actions and including extra transitions[2]. The *Sink* in Figure 5 is a good illustration of the approach. Specifically, rather than refering to actions *sinkin* and *play* as was the case in our earlier formulations of the stream, now it refers to actions *sinkin*0?, *sinkin*1?, *play*0? and *play*1?. Thus, we have flattened out our data type into a more discriminating set of action names.

The formula that we would like to verify over this automata is:

$$\forall x \in \{0,1\} \,.\, (\Box(\mathit{play}(x) \Rightarrow \Diamond\!\!\!\!\!{-}\,_{\leq 95}\, \mathit{sourceout}(x))) \qquad (*)$$

Unfortunately a further problem remains: UPPAAL does not support past operators[3]. We could though reformulate the property as:

$$\forall x \in \{0,1\} \,.\, (\Box(\mathit{sourceout}(x) \Rightarrow \Diamond_{\leq 95}\, (\mathit{play}(x) \ \vee \ \mathit{loss}(x)))) \qquad (*)$$

which avoids the past operator. However we prefer an alternative approach that avoids the reference to *loss*. This is because latency is an end-to-end property and formulating in terms of actions local to components of the communication path seems conceptually unsatisfactory. Thus, we would like to view the medium as a black box and formulate our property purely in terms of the "end-point" actions *sourceout* and *play*.

In order to do this, let us consider the interplay between loss and latency. In the presence of congestion, loss will relieve congestion and thus allowing loss will implicitly reduce latency values. Thus, a reasonable strategy is to determine upper bounds on latency on a stream specification which does not allow loss, knowing that if it is added, this bound will still be valid. This is the strategy that we adopt.

So, let us work with a basic medium which does not contain the possibility to loose packets. The medium is as shown in Figure 5.

Now the property that we have to check is:

$$\forall x \in \{0,1\} \,.\, (\Box(\mathit{sourceout}(x) \Rightarrow \Diamond_{\leq 95}\, \mathit{play}(x)))$$

[2] This is in fact a standard approach in process algebras for getting from a data passing calculus to a basic calculus, see for example [14].

[3] Actually, there is in any case a rather subtle problem with this formula, to do with the interplay between the possibility to loose messages and not knowing the end-to-end latency.

which can, in the standard way, be expanded out to avoid the universal quantifier, which is not supported in UPPAAL (note: we write $a(n)$ as an in order to match action denotations in the automaton).

$$\Box(sourceout0 \Rightarrow \Diamond_{\leq 95}\, play0) \ \wedge \ \Box(sourceout1 \Rightarrow \Diamond_{\leq 95}\, play1)$$

However, this is not a reachability property and can thus, not be directly verified using UPPAAL. A strategy outlined in [12] can though be used to verify such a "bounded liveness" property using reachability analysis. The approach is to derive a testing automaton from the property using a form of tableau algorithm, compose the testing automaton in parallel with the system and verify a simple reachability property. This strategy is not yet implemented in the UPPAAL tool, so the automaton has been derived by hand using the informal algorithms to be found in the literature [12].

The bounded liveness properties accepted by the testing automaton approach are expressed in a different logic to any that we have seen so far: STL, a timed modal logic. Our property can, with relative ease, be expressed in this logic however, rather than introduce another logic, we will go straight to the testing automaton that is derived from the formula. It is shown in Figure 5 along with the full revised stream specification. In fact, this is the scenario used to check that *sourceout*0s and *play*0s are correctly matched. A similar approach can be used to check *sourceout*1's and *play*1's.

Note that *sgout*0 is a probe action that has been inserted at appropriate places in the system. It is inserted (using a committed state) to signal the occurrence of *sourceout*s to the test automaton.

The property that we check is:

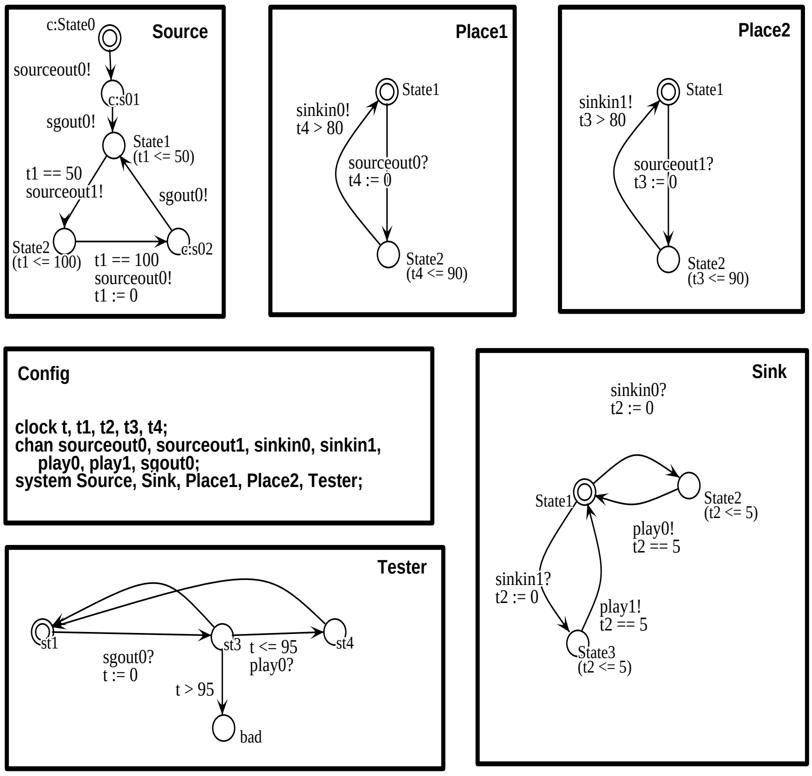
$$E <> (Tester.bad)$$

which, when checked with UPPAAL does, as would be hoped, fail to hold. However as indicated earlier, this is not a very interesting result because it is directly deducible by inspection of the system. Thus, the contribution of this section is not this verification, but rather the investigation of a general strategy for checking latency which can be applied to systems that are not so easily interpreted. It is clear that the strategy we have documented is indeed generally applicable.

6 Concluding Remarks

We have investigated the suitability of UPPAAL for the verification of multimedia systems. The specification and analysis of a simple multimedia stream was presented for this purpose. The main results of this paper are the identification of generally applicable strategies for checking real-time quality of service properties, specifically, checking throughput and latency.

Although our experiences with UPPAAL have generally been favourable, some criticisms of the approach can be highlighted. [6] considers a number of



```
clock t, t1, t2, t3, t4;
chan sourceout0, sourceout1, sinkin0, sinkin1,
    play0, play1, sgout0;
system Source, Sink, Place1, Place2, Tester;
```

Fig. 5. The Stream with Testing Automata

such criticisms. One criticism is though particularly worth considering here as it arises directly from our case study. It is that due to expressiveness limitations of the temporal logic accepted by the UPPAAL model checker it is difficult to directly verify the standard temporal logic formulations of quality of service, rather the basic system specification has to be adapted in order that checking the property can be reduced to checking a reachability property. This can most noticeably be seen in the latency verification where the basic system specification has to be adapted through composition of a test automata and addition of probe actions. Consequently the verification is not "transparent" to the behavioural specification.

Other real-time model checking tools, in particular KRONOS [8], support a richer set of temporal logic properties. Thus, we hope that verification that avoids such invasive adaptation of the basic system specification may be possible with KRONOS. Ongoing research is investigating application of KRONOS to the multimedia stream case study.

Acknowledgements

We would like to thank Stavros Tripakis of SPECTRE-VERIMAG and Paul Pettersson of UPPSALA who fielded queries that we had on model checking of timed automata and UPPAAL respectively.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, (126):183–235, 1994.
2. V. Bellotti and A. MacLean. Integrating and communicating design perspectives with QOC design rationale. Technical Report ID/WP29, ESPRIT 7040 - AMODEUS, 1994.
3. 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.
4. G.S. Blair, L. Blair, H. Bowman, and A. Chetwynd. *Formal Specification of Distributed Multimedia Systems.* University College London Press, September 1997.
5. M. Bordegoni, G. Faconti, S.Feiner, M.Maybury, T. Rist, S. Ruggieri, P. Trahanias, and M. Wilson. A standard reference model for intelligent presentation systems. *Computer Standards and Interfaces*, 1998.
6. H. Bowman, G. Faconti, J-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronisation algorithm using UPPAAL. *Accepted at FMICS'98*, Amsterdam, The Netherlands, May 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. P.G. Harrison and N.M. Patel. *Performance Modelling of Communication Networks and Computer Architectures.* Addison-Wesley, 1993.
10. Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 2–13, San Francisco, California, USA, 3-5 December 1997.
11. I. Herman, G. Reynolds, and J. Van Loo. PREMO: An emerging standard for multimedia. part i: Overview and framework. *IEEE MultiMedia*, 3:83–89, 1996.
12. 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.
13. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.
14. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
15. 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.