UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE

38100 Povo – Trento (Italy), Via Sommarive 14
http://www.disi.unitn.it

Modeling a distributed Heterogeneous Communication System using Parametric Timed Automata

Thi Thieu Hoa Le, Luigi Palopoli, Roberto Passerone and Yusi Ramadian
April 2010

Technical Report Number: DISI-10-031

Modeling a distributed Heterogeneous Communication System using Parametric Timed Automata

Thi Thieu Hoa Le, Luigi Palopoli, Roberto Passerone, Yusi Ramadian
Department of Information Engineering and Computer Science
University of Trento, Italy
hoa.le@studenti.unitn.it

April 20, 2010

Abstract

In this report, we study the application of the Parametric Timed Automata(PTA) tool to a concrete case of a distributed Heterogeneous Communication System (HCS). The description and requirements of HCS are presented and the system modeling is explained carefully. The system models are developed in UPPAAL and validated by different test cases. Part of the system models are then converted into parametric timed automata and the schedulability checking is run to produce the schedulability regions.

1 Introduction

Symbolically computing the region in the parameters space that guarantees a feasible schedule (given a set of real-time tasks characterised by a set of parameters and activation patterns) is a novel approach to the computation of schedulability regions[2]. This method is of great usefulness, for example, once the feasible regions have been identified, the system designer can choose quickly a correct set of parameters that could make the system works properly. Moreover, he can also be assisted in optimizing the system performance while still keeping the system schedulable.

Parametric timed automata have been approached differently in the literature. In [7] given a real-time system and some temporal formula which may contain parameters, and a constraint over the parameters, a model-checking problem is to verify whether every allowed parameter assignment could guarantee that the real-time system satisfies the formula. Instead, in [2] sensitivity constraints over the interested parameters are computed and then joined together to produce the schedulability region of the system.

Another interesting work on PTA is studied in [8] where the authors use a prototype extension of UPPAAL [4] for linear parametric model checking and show decidability results for the verification of *L/U automata*. For this special class, the problem of deciding whether there exists a parameter valuation such that a given location s is reachable, is indeed decidable while it is not for the full class. The idea to this emptiness problem is then generalized in [2] in order to produce the parameter region in which the system is unschedulable.

In the context of asynchronous circuits, E. Andre et al. [9] propose a method of synthesizing constraints of a timed automaton, given an initial set of parameter values for which the system is known to behave properly. The authors ensure that for any two valuations of parameters satisfying these constraints, the behaviors of the timed automata are time-abstract equivalent. Although the method will terminate as long as all symbolic traces computed from a given *reference parameter valuation* are either of finite length or trivially cyclic, it has been shown to be particularly suitable in the framework of asynchronous circuits. A. Cimatti et al.'s solution to the synthesis of constraints [2] is also symbolic but but differs from the former approach in many aspects. The later aims at symbolically computing the *region of parameter space* that makes the system feasible by enumerating all possible traces that could drive the system into an error state and identifying, for each of them, the subsets of the parameter space that are compatible with the trace. In addition, the method *does not* make use of reference parameter valuations

and it is proved to converge for periodic task systems with bounded offsets.

In fact, the method in [2] can be applied widely in real-time systems adopting a fixed priority mechanism and in an effort to help with the development of the PTA prototype tool, we have tried to apply the tool to an industrial embedded application. The application is first modeled in UPPAAL and validated by different ground sets of parameters. Part of the system models are then converted into parametric timed automata which are then analyzed by the PTA tool in order to produce the feasible region.

This report is organized as follows. Section 2 describes the application and its requirements. In section 3, the UPPAAL models for the application are explained in detail. These models are then validated by verifying different sets of parameters in section 4. Next, section 5 presents the parametric analysis after running the PTA tool. Finally, section 6 concludes the report and suggests future work.

2 HCS and Parametric Timed Automata

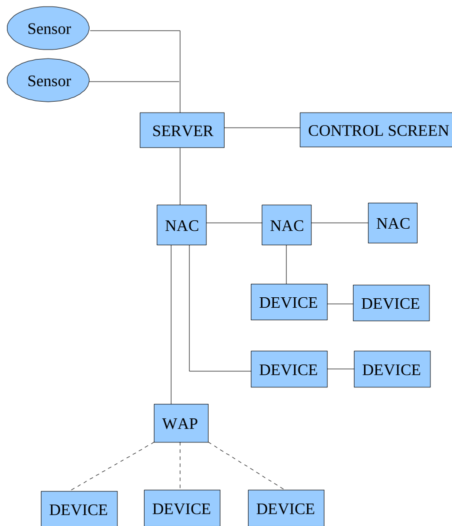


Figure 1: Heterogeneous Communication System(HCS)

The distributed Heterogeneous Communication System (HCS) contains various devices, wired and wireless communication networks and a common server. The HCS provide control, monitoring and data processing of various subcomponents through heterogeneous networks. The server is connected to different devices such as sensors and actuators via wired and wireless protocols. The various devices are connected to the server through Network Access Controllers (NAC). The architecture of the HCS is described in more detail in [1].

HCS provides two important applications that can be deployed on end devices. One is to transmit audio data periodically to end devices every audPeriod ms from the server. The other application is to synchronize clocks using the Precision Time Protocol (PTP, IEEE1588[5]).

A general HCS is depicted in figure 1, based on wired and wireless components. The HCS system consists of the following components: SERVER, DEVICE, NACs, wired and wireless networks. The HCS server is connected to all NACs in a daisy-chain topology. The NACs perform the gateway function between the backbone and the end devices. Wireless devices are accessed via wireless access point (WAPs), and WAPs are connected directly to NACs. Functions of the WAPs are similar to the NACs functions, in particular synchronization with the network and server, data routing between NACs and wireless network.

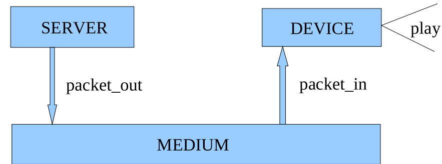


Figure 2: General schema of audio streaming over a network

Figure 2 illustrates a common design approach for audio streaming. More detail can be found in [1]. Such a stream of the HCS has a number of noticeable characteristics as follows:

- The communication between the server and device is asynchronous.
- The server sends an audio packet every $audPeriod$ ms.
- An audio packet is characterized by two parameters: a sequence number i and a timestamp ti denoting the time the packet has to be played at the device.

- Packets arrive at the device (except they get lost) with a minimal latency $Lmdm = Lmin$ and a maximal latency $Lmdm = Lmax$
- The NACs simply forward the incoming packets to the devices. And packets passing through the NACs have to experience a further delay of $Lnac$ ms during which they are preprocessed by the NACs.
- When a packet is received by the device, it needs τ ms to process the packet after which it is ready to receive the next packet.
- The medium is unreliable, it may lose and reorder packets.

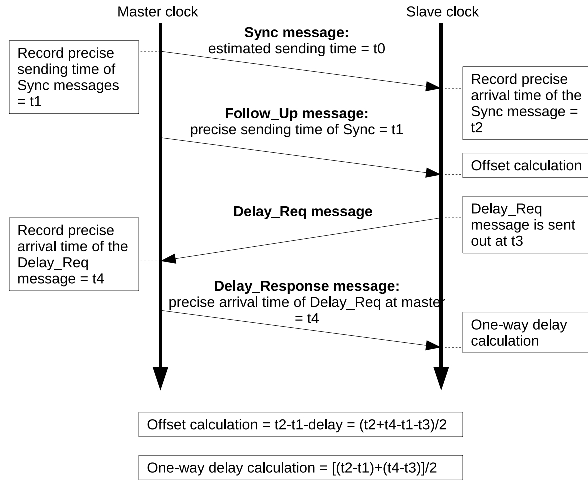


Figure 3: Time sequence diagram of message exchanged between master and slave clock to achieve time synchronization

Figure 3 depicts the clock synchronization of the slave clock with the master clock based on PTP. Every component of HCS has a local clock and PTP runs in the server, devices and NACs. Various timing delays are to be guaranteed, for example, in a scenario where two devices are connected to the server both wired and wireless, it should be guaranteed that both devices are synchronized within an error of 0.1ms (synchronization precision).

The ideal objective is to identify the parameter space-the largest region in which the correct functioning of audio streaming and clock synchronization can be guaranteed by using the novel method proposed in [2]. One requirement regarding the correct functioning of the audio application, for instance, could be the maximal time difference between sending an audio packet and playback of the audio packet at the devices is less than 0.1ms. Another synchronization requirement could be to ensure the synchronization precision is bounded by 0.1ms, given different timings on wired and wireless network.

However, HCS is too complex to be parametrically-modeled completely. Therefore, before applying the parametric timed automata (PTA) approach, we need to relax some of the above requirements. The simplified system would contain only one server and many NACs and devices where one NAC may be associated to at most one device and one other NAC. Also, for the time being we would just focus on modelling the PTP part on the server and devices. Additionally, every audio packet will have to experience a maximal delay when traversing through the medium ($Lmdm = Lmax$). Furthermore, the last two characteristics of the HCS audio streaming and the timing PTP delay are temporarily not considered and would be in the future as the next modelling step. Lastly, assuming that the transmission priority of the packets is as follows: $Prior_{Sync} > Prior_{Follow_Up} > Prior_{Delay_Res} > Prior_{Delay_Req} > Prior_{Audio}$. The system high level description can be viewed logically as in figure 4(a) or figure 4(b).

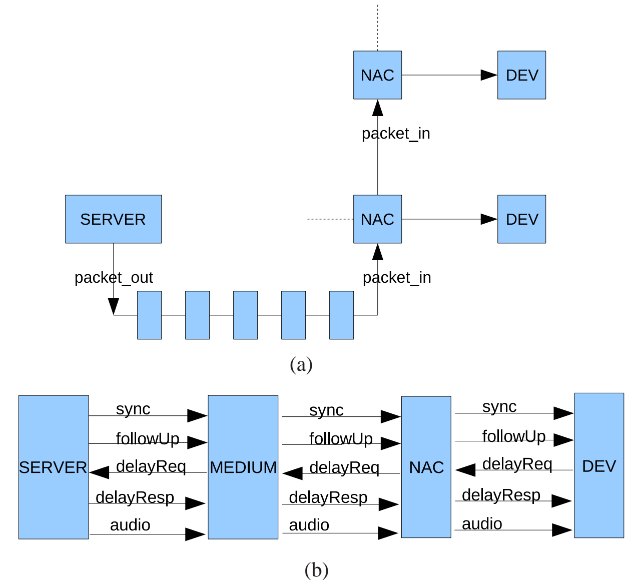


Figure 4: Logical model of HCS

3 UPPAAL Models

The models of HCS are first developed in UPPAAL[4] because UPPAAL allows graphically-modeling ability which assists model-developers in debugging and testing their models.

3.1 Integer clock

In UPPAAL, HCS is modeled as a network of extended timed automata with global real-valued clocks and integer variables. Since the clock value is necessary in the PTP protocol, we need to retrieve the clock value which is impossible in UPPAAL as it does not support real variables as well as clock value retrieval. And integer clocks are invented to overcome this hurdle. In fact, an integer clock is an integer variable returning the integer part of a real-valued clock. This applies to real clocks which have no clock drift. However, since the local clocks in the server and devices usually drift compared to the actual time, we need to adjust the operation of integer clocks in order to retrieve the clock value within a certain precision. Assuming we have a clock drift e $(0 < e)$, a real clock c and an integer clock ci. Actually, e should be a real variable but as this is not allowed in UPPAAL, e has to be integerized. That is, if we want to take care of up to n digits after the decimal point, we multiply e and 10^n together. For example with $n = 2$, $e = 95$ means the drift of 0.95 and $c = 0.95 * t$ where t is not a drifting clock. Thus, ci increases by 1 when $t = 100/e = 100/95 = 1.052631579$. Again, UPPAAL does not allow comparisons of clocks with real values, so we have to integerize $100/e$ and make it even more precise by multiplying it by some precision $prec$. With a three-digit precision $prec = 1000$, for instance, ci increases when $t = 1000 * 100/95 = 1052$, that is we are scaling the bound at which the integer clock is changed. Hence ci is more precise. Furthermore, every integer variable will finally overflow if it keeps increasing, therefore to ensure the correctness of the whole system, it is necessary to reset integer clocks to 0 whenever they reach a predefined clock limit $clkLimit$.

3.2 Server

The server is modeled as a network of five timed automata, one of which models the server integer clock, two of which model the PTP protocol running in the server and the others model the audio sending and buffering operations of the server.

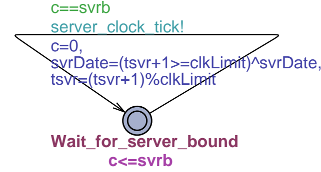


Figure 5: The server integer clock

The timed automaton modeling the server integer clock is shown in figure 5. At the beginning the clock bound of the server is $svrb = prec * 100/esvr$ where $prec$ is the digit precision and $esvr$ is the server clock drift. When clock c reaches the bound, the server integer clock $tsvr$ will increase by 1 and c is reset to 0. As said above, to prevent the overflow situation, when $tsvr$ reaches the clock limit $clkLimit$, it will be reset to 0. Because the server integer clock is reset every $clkLimit$ time units and so is the device integer clock, it is crucial to distinguish the states of two clocks, that is whether they have been reset or not. Given that the difference of the two clocks can never exceed $clkLimit$, we invent the notion of "odd date" and "even date". The clocks at the beginning show the time in "even date" and when they reach the clock limit, the displayed dates are changed to "odd date" and vice versa. The server date is encoded in the variable $svrDate$ which results from the binary operation XOR: $svrDate = (tsvr + 1 >= clkLimit)\ XOR\ svrDate$.

The PTP timed automata are depicted in figure 6. Figure 6(a) illustrates the first two steps of the synchronization procedure shown in figure 3 and figure 6(b) describes the last step. In the figures, all edges are normal channels used to synchronize two timed automata except $sync_released$ and $delayRsp_released$ - two broadcast channels that can always fire (provided that the guard is satisfied), no matter if any receiving edges are enabled. But those receiving edges, which are enabled, will synchronize.

In figure 6(a), the sync-release task has the period of $syncPeriod$ and may have some initial offset $syncOff$. After the sync-release event is activated, the Sync message must be delivered to the medium within $halfDelta * svrb$ time units. The time when the Sync has been transmitted completely onto the medium should be recorded so that it will be added to the Fol-

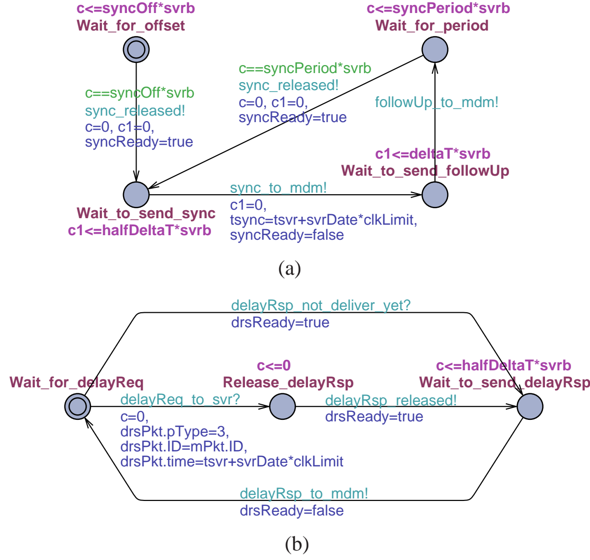


Figure 6: The PTP protocol running in server

low_Up message and sent to the devices: $tsync = tsvr + svrDate * clkLimit$. Here we encode the server date and time information into the Sync sending time. Similarly, after at most $deltaT * svrb$ time units since the Sync transmission, the Follow_Up message has to be delivered to the medium, $halfDeltaT$ and $deltaT$ are defined in the PTP protocol[5].

In figure 6(b), the timed automaton initially waits for Delay_Req messages from the devices. When one such message arrives, it prepares a Delay_Res message to send back to the device:

- $drsPkt.pType = 3$: the message is Delay_Res.
- $drsPkt.ID = mPkt.ID$: the destination is the source of the Delay_Req message and $mPkt$ is the packet the server received from the medium.
- $drsPkt.time = tsvr + svrDate * clkLimit$: encoding the server date and time information.

Again, the Delay_Res must be delivered to the medium within $halfDeltaT$ time units after the reception of the Delay_Req message. However, if higher priority messages are available at its delivery time, the Delay_Res will backoff ($delayRsp_not_deliver_yet$) and retry after a short time.

Figure 7(a) describes the audio sending operation of the server assuming that audio packets are periodically generated and played. The audio-release task has the period of $audPeriod$ and probably some initial offset $audOff$ and also a relative deadline $relD$ at which it must be played. When the task is activated, an audio packet will be either delivered to the medium or pushed into a buffer depending on whether the medium is busy or not. The audio packet will contain the time the packet has to be played at the devices ti. The procedure $audPkt$ helps with this preparation.

Procedure 1 audPkt(int deadline, int &ti, int &audDate)

```
1: audDate=(deadline>=clkLimit) XOR audDate;
2: ti=deadline % clkLimit;
```

Figure 7(b) describes the audio buffering operation of the server. The timed automaton will buffer or remove an audio packet by taking the $audio_to_buf$ or $audio_from_buf$ transition respectively. When a buffer over-run occurs, the automaton goes to the Error state. Moreover, two auxiliary procedures are used to simplify the automata. The $audPush$ procedure helps to encode the date and time information into the audio sending time and push waiting packets into a buffer. The $audPop$ procedure helps to remove a packet from a buffer. It is also noticeable that $lookAhead$ always points to the first element of the buffer when it is not empty. In addition, $audio_to_buf$ is modeled as a broadcast channel so that audio packets are buffered when the medium is busy.

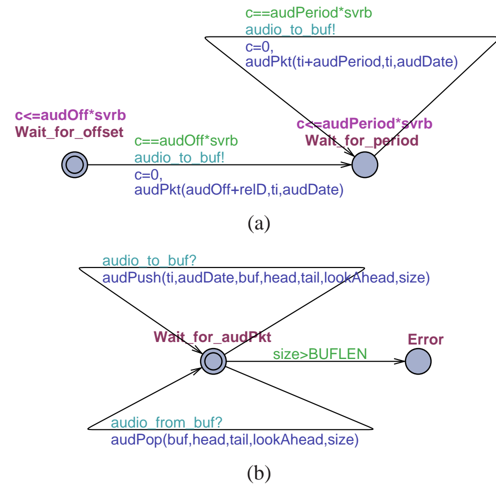


Figure 7: The audio sender and buffer of server

3.3 Medium

The medium is responsible for data transmission, so whenever there is available data waiting to be transmitted, the transmission must take place unless the medium is currently busy. In a heterogeneous system, the transmission decision is more complex as different packets have different priorities. Only the packet with the highest priority would be selected to be transmitted, others will have to backoff and wait for their turn. Figure 8(a) shows a simplified timed automaton modeling the transmission decision when many packets are available at the same time. In this figure, $sync_released$, $delayRsp_released$, $audio_to_buf$, $delayReq_to_NACOutBuf$, $pkt_to_NACInBuf$ are modeled as broadcast channels, or more precisely, receiving edges which will synchronize with the sending edges whose name is exactly the same. Initially, the automaton can choose nondeterministically one receiving edge to take.

- If $sync_released$ was selected, the medium would not have to care about other packets as Sync has the highest priority.
- If $delayRsp_released$ was selected, the medium would look for any sign of sync-release. If there is not, it would allow the Delay_Res to be delivered to the medium. However, if the sync-release happens before it could actually start the transmission ($c = 0$), the Delay_Res will backoff and give way to the Sync transmission.
- If $delayReq_to_NACOutBuf$ was selected, again the medium would look for signs of packets with higher priorities. The Delay_Req would backoff and give way to any such packet if ready.
- The similar situation happens when $audio_to_buf$ was selected.

When the transmission decision has already been decided, the medium transmits the packet for $Lmdm * mdmb$ time units where $mdmb = prec$ is the clock bound of the medium, and depending on the packet type the medium can take different actions upon completing its transmission:

- $pType = 1$ (Sync): the medium puts the packet into the NAC input buffer and starts transmitting the Follow_Up because it has the second highest priority.

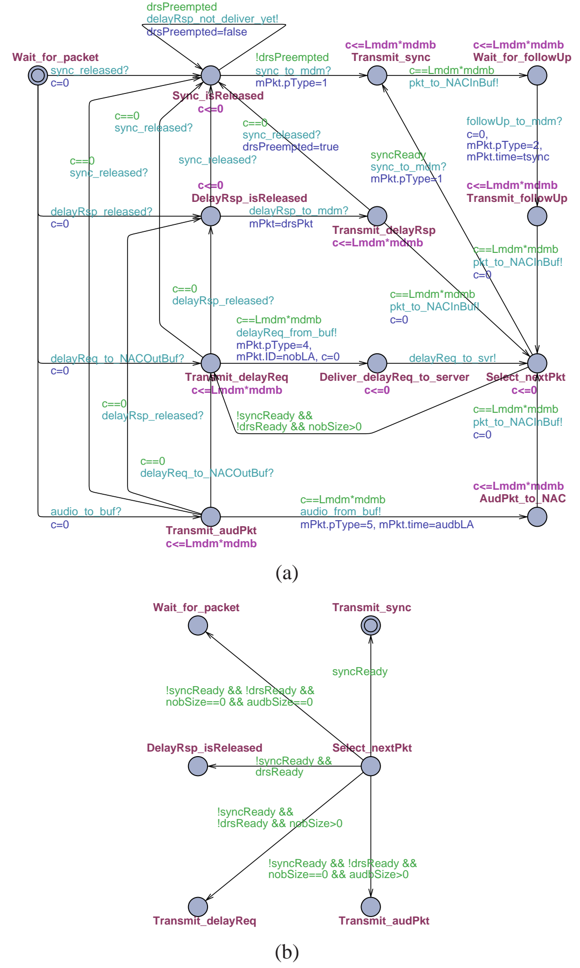


Figure 8: A simplified automaton of the medium

- $pType = 3$ (Delay_Res): the medium puts the packet to the NAC input buffer.
- $pType = 4$ (Delay_Req): the medium confirms its transmission so that the packet is safely removed from the NAC output buffer and delivered to the server.
- $pType = 5$ (Audio): likewise, the audio packet is removed from the audio buffer and put into the NAC input buffer.

The NAC input and output buffer will be discussed later. For every buffer, there is a look ahead variable that always points at the first element of the buffer when it is not empty, such as $nobLA$ and $audbLA$ - the look

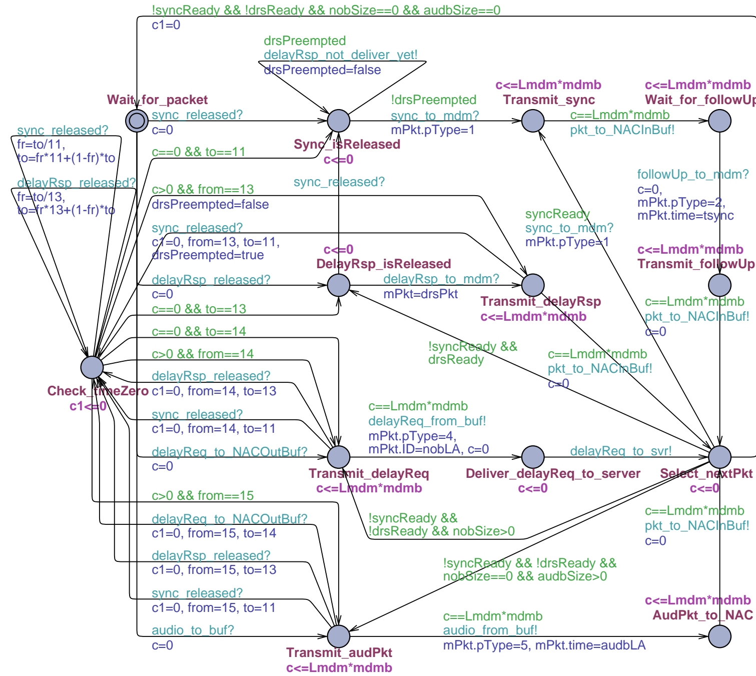


Figure 9: A complete automaton of the medium

ahead variables of the NAC output buffer and the audio buffer respectively.

Now that the automaton is selecting the next packet to transmit, if:

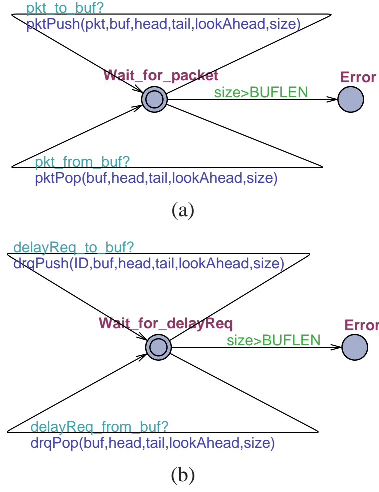
- *syncReady* (Sync is ready): the medium starts the Sync transmission.
- *!syncReady && drsReady* (Delay_Res is ready): the medium starts the Delay_Res transmission.
- *!syncReady && !drsReady && (nobSize>0)* (Sync and Delay_Res not ready and the NAC output buffer not empty): the medium starts the Delay_Req transmission.
- *!syncReady && !drsReady && (nobSize==0) && (audbSize>0)* (only audio packets are ready): the medium starts the Audio transmission.
- *!syncReady && !drsReady && (nobSize==0) && (audbSize==0)* (no packet is available): the medium goes back to the initial state.

Figure 8(b) depicts these next packet selections. The complete medium automaton would be obtained by joining the states and edges in figure 8(a) and figure 8(b). However, since UPPAAL does not allow clock guards on receiving edges of broadcast channels, we have to add one more state to check the time at which the preemption happens. Figure 9 shows the complete medium automaton.

3.4 Network Access Controllers (NACs)

The NACs is responsible for data routing from the server into subnet(s) and vice versa. Also, the NACs can perform data encryption/decryption on every packet passing through it. Because only one packet can be processed at a time, NACs are assumed to have two buffers to contain incoming or outgoing packets. The NAC input buffer contains packets coming only from the medium and the NAC output buffer contains only Delay_Req packets going from the devices to the medium. These buffers are depicted in figure 10.

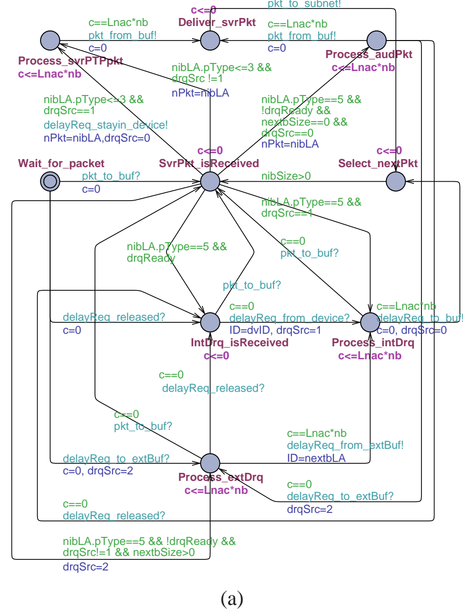
The input automaton will add or remove a packet by taking the $packet_to_buf$ or $packet_from_buf$ transition respectively. Similarly, the output automaton will add or remove a Delay_Req message by taking the $delayReq_to_buf$ or $delayReq_from_buf$ respectively. In this figure, pkt denotes the incoming packet and ID the identity of the device sending the Delay_Req. When a buffer over-run occurs, the corre-



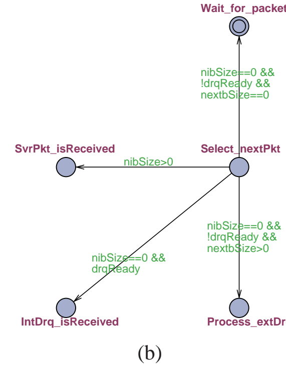
(a)

(b)

Figure 10: The NAC input and output buffer



(a)



(b)

Figure 11: A simplified automaton of the NACs

sponding automaton goes to the Error state.

Similar to the medium case, the NACs always transmit packets of the highest priority among the ready packets. Figure 11 shows a simplified timed automaton modeling the activities of a NAC. In this figure, pkt_to_buf, $delayReq_released$, $delayReq_to_extBuf$ are receiving edges of broadcast channels while $delayReq_to_buf$, pkt_to_subnet are emitting edges of broadcast channels. Initially, the automaton can choose nondeterministically one receiving edge to take.

- If pkt_to_buf was selected, the NAC looks further at the packet type. If it was either a Sync, Follow_Up or Delay_Res message ($pType <= 3$), the NAC ignores other packets and goes on processing the current packet. In case the Delay_Req from the device is preempted by the current packet, the NAC tells its attached device to backoff the Delay_Req by synchronizing on the channel $delayReq_stayin_device$. On the contrary, if it was an Audio packet then the NAC would still ignore other packets as long as no Delay_Req from the attached device is ready (*drqReady==true*) or preempted (*drqSrc==1*) and no Delay_Req from external NACs is ready (*nextbSize==0*). Assuming that the Delay_Req from the attached device has a higher priority than that from external NACs, so when they are ready at the same time the former preempts the latter.
- If $delayReq_released$ was selected, the NAC would look for any sign of incoming packets. If there is not, it would process the Delay_Req from its attached device. Otherwise, only when the incoming packet has a higher priority will the Delay_Req backoff.
- If $delayReq_to_extBuf$ was selected (another Delay_Req is coming from an external NAC output buffer), again the NAC would look for signs of packets with higher priorities. The Delay_Req would backoff and give way to any such packet if it is available.

When the NAC has made its decision, it will process the packet for $Lnac * nb$ time units where $nb = prec$

is the clock bound of the NAC. After that, the NAC can take different actions upon completing processing, depending on whether the packet is incoming or outgoing:

- For incoming packets: the NAC confirms its input buffer that the packet has been processed successfully so that the packet can be safely removed from the input buffer by synchronizing on the pkt_from_buf channel. The packet is then forwarded to the subnets (including external NACs and devices).

- For outgoing packets: if it was an external Delay_Req from an external source, the NAC needs to confirm the external buffer from which the Delay_Req is safely removed. After that, the Delay_Req is put into the NAC output buffer and waiting for the medium to take it. In addition, the identity of the device ($dvID$) should be included inside the Delay_Req message so that when it is received, the server could send back a Delay_Res.

The look ahead variables for the NAC internal input, internal output and external output buffer are $nibLA$, $nobLA$ and $nextbLA$ respectively.

Now that the automaton is selecting the next packet to process, if:

- *nibSize>0* (the NAC input buffer is not empty): the NAC tries processing incoming packets.

- *(nibSize==0) && drqReady* (the NAC input buffer is empty but a Delay_Req from its attached device is ready): the NAC takes the Delay_Req in.

- *(nibSize==0) && !drqReady && nextbSize>0* (only Delay_Req from external NAC output buffer is ready): the medium tries processing the external Delay_Req.

- *(nibSize==0) && !drqReady && (nextbSize==0)* (nothing to process): the NAC goes back to its idle state and waits for new packets.

Figure 11(b) depicts these selections. The complete NAC automaton would be obtained by joining the states and edges in figure 11(a) and figure 11(b). However, as in the medium case, since UPPAAL does not allow clock guards on receiving edges of broadcast channels, we have to add one more state to check the time at which the preemption happens. Figure 12 shows the complete NAC automaton.

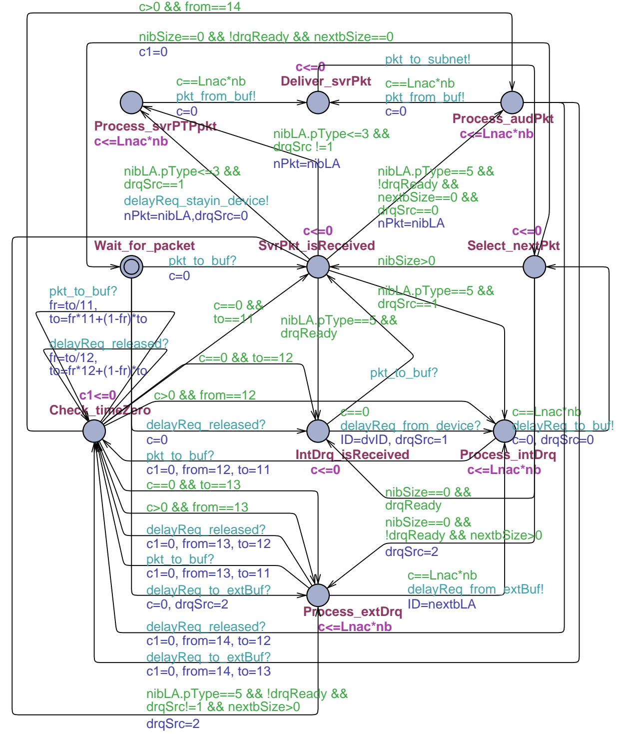


Figure 12: A complete automaton of the NACs

3.5 Device

The device is modeled as a network of four timed automata, one of which models the device integer clock, two of which model the PTP protocol running in the device and the other models the audio receiver.

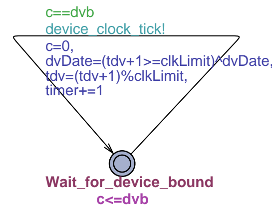


Figure 13: The integer clock of devices

The timed automaton modeling the device integer clock is shown in figure 13. At the beginning the clock bound of the device is $dvb = prec * 100/edv$ where $prec$ is the digit precision and edv is the device clock drift. When clock c reaches the bound, the device integer clock tdv will increase by 1 and c is reset to 0. As done for the server integer clock, when tdv

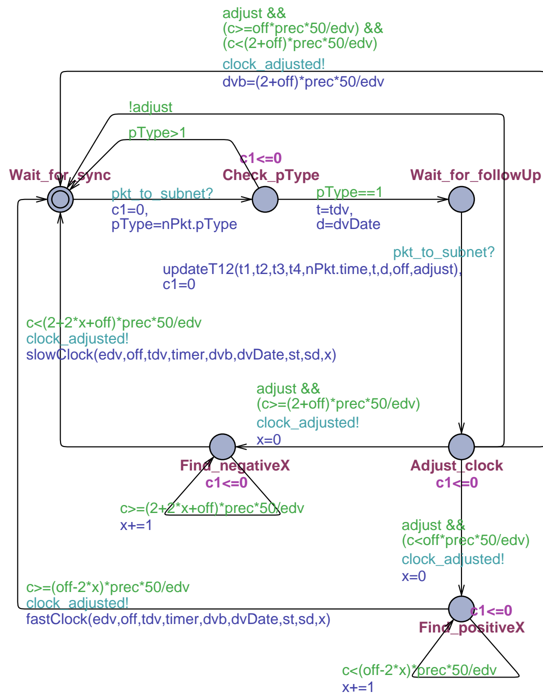


Figure 14: The first two steps of the PTP protocol running in devices

reaches the clock limit $clkLimit$, it will be reset to 0. Also the device date is encoded in the variable $dvDate$ which results from the binary operation XOR: $dvDate = (tdv + 1 >= clkLimit)\ XOR\ dvDate$. In addition, there is a $timer$ keeps increasing in the figure. We will see later how this variable is used.

The first two steps of the PTP protocol running in the device are depicted in figure 14. The device waits until a Sync message arrives and records the arrival date and time of the Sync. Then when a Follow_Up comes, it checks whether the Sync sending and receiving actions happen in a same day (this is a constraint added to simplify the modeling of PTP). If it is the case, the device further checks whether the slave-to-server delay is available ($t4 > 0$). Then if such delay is not available, the device goes back to its initial state. Otherwise, it proceeds with the clock adjustment. The auxiliary procedures *updateT12* helps the device in making its adjustment decision.

The current scaled time point is $tdv * prec + edv * c/100$ (tdv and c are two shared variables of the integer clock timed automaton), after clock adjusting it becomes $tdv * prec + edv * c/100 - off * prec/2 =$

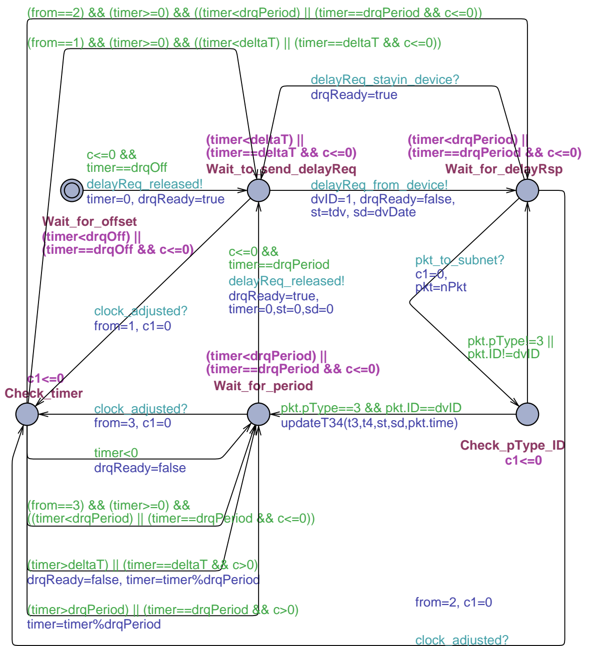


Figure 15: The last two steps of the PTP protocol running in devices

Procedure 2 updateT12(int &t1, int &t2, int &t3, int &t4, int &tsend, int &trecv, int &recvDate, int &off, bool &adjust)

1: int sendDate=(tsend>=clkLimit);
2: adjust=false;
3: **if** sendDate==recvDate **then**
4: t1=tsend-sendDate*clkLimit;
5: t2=trecv;
6: **if** t4>0 **then**
7: adjust = true;
8: **end if**
9: **if** adjust **then**
10: off+=t2+t3-t1-t4;
11: **end if**
12: **end if**
13: trecv=0;
14: recvDate=0;

$tdv_{new} * prec + delta$ where $0 <= delta < prec$. There are three possibilities:

- $0 <= (edv * c/100 - off * prec/2) < prec$ or $(off * prec * 50/edv) <= c < [(2 + off) * prec * 50/edv] \Rightarrow tdv_{new} = tdv$ and $dvb_{new} = [(2 + off) * prec * 50/edv]$.

- $(edv * c/100 - off * prec/2) < 0$ or $c < (off * prec*50/edv) \Rightarrow$ the device clock is running faster than the server clock and so there exists $x > 0$ such that $0 <= tdv * prec + edv * c/100 - off * prec/2 + x * prec < prec$ and $tdv_{new} = tdv - x$ and $dvb_{new} = [(2 - 2 * x + off) * prec * 50/edv]$

- $(edv * c/100 - off * prec/2) >= prec$ or $c >= [(2 + off) * prec * 50/edv] \Rightarrow$ the device clock is running slower than the server clock and so there exists $x > 0$ such that $0 <= tdv * prec + edv * c/100 - off * prec/2 - x * prec < prec$ and $tdv_{new} = tdv + x$ and $dvb_{new} = [(2 + 2 * x + off) * prec * 50/edv]$

Moreover, the delayReq-release task has the period of $drqPeriod$ and may have some initial offset $drqOff$. After the delayReq-release event is activated, the Delay_Req message must be delivered to the medium within $deltaT * dvb$ time units. The time when the Delay_Req has been taken in by the NAC should be recorded so that it will be used later in the PTP protocol.

Figure 15 shows the last two steps of the synchronization procedure in the devices. In this figure, all edges are normal channels except $delayReq_released$ and $clock_adjusted$ - two broadcast channels.

After at most $drqPeriod * dvb$ time units since the Delay_Req delivery, the Delay_Res message has to be received by the device according to the PTP protocol. These timing constraints are enabled by using a timer. This is because the device clock bound is changed periodically and not static as the server clock bound. Additionally, the timer is put forward or backward the same amount of time as the local clock is. So after the clock adjustment, if the timer does not respect the invariants any longer, the automaton will abort the activity it was taking before the clock adjustment took place. For example, if *(timer<0)* or *(timer>deltaT)* or *((timer==deltaT) && (c>0))* (c is a shared variable of the integer clock timed automaton), it cancels the current delivery and waits to start a new Delay_Req delivery. If no invariant violations is committed, the automaton executes its normal cycle, that is sending a Delay_Req to the server, receiving the corresponding Delay_Res and updating the slave-to-master delay information. The auxiliary procedure *updateT34* helps with the last step in the cycle, assuming that the Delay_Req sending and Delay_Res receiving actions should happen in a same day (this is another constraint added to simplify the modeling of PTP).

Procedure 3 updateT34(int &t3, int &t4, int &tsend, int &sendDate, int &trecv)

1: int recvDate=(trecv>=clkLimit);
2: **if** sendDate==recvDate **then**
3: t3=tsend;
4: t4=trecv-recvDate*clkLimit;
5: **end if**

Lastly, figure 16 describes the audio receiving operation of the device. The date and time information of the audio packet is retrieved and checked with the current date and time of the device local clock. If the clock has passed the time at which the packet must be played, the automaton will go to the Error state.

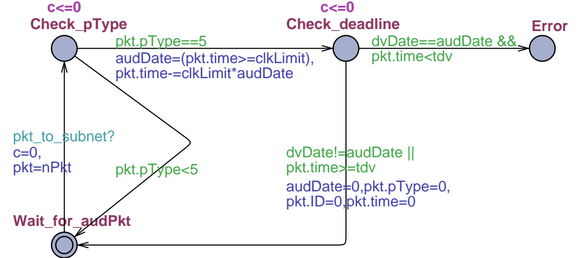


Figure 16: The audio receiver of devices

4 Ground Verification

The above models have been validated by UPPAAL, a model-checker for timed-automata [4]. In the previous section, all of the models were constructed in the syntax of UPPAAL. Thus, it is trivial to feed them directly to UPPAAL so that their properties can be checked by the verifier of the tool. UPPAAL uses a timed Computation Tree Logic (CTL) language for specifying properties which we want to verify. To verify that the system is schedulable, we must show that the four error states are never reachable. Using the timed CTL language, the schedulability properties are specified as follows:

- A[]!SVR_AudBuf.Error
- A[]!NACInBuf.Error
- A[]!NACOutBuf.Error
- A[]!DV_AudRecvr.Error

Only if these schedulability properties are satisfied, can we be sure that the system is schedulable. If just one property is violated, the system is not guaranteed to work properly because it may encounter the deadlock state where none of the timed automata can move. Several test cases are performed to illustrate this point. In the first test case, the schedulability of the system is guaranteed because all properties are satisfied while it is not in the second case due to the violation of the last property. The last test case points out the possibility of deadlock when the second property is not satisfied. In all test cases, the system consists of one server, one medium, one NAC and one device.

4.1 Test case 1

In this test case, the Sync interval and the period of the Delay_Req message are equal to 20ms while that of the Audio packets is 40ms. The offset parameters are 0 for the sync and audio packets and 10ms for the Delay_Req message. Moreover, the latency parameters of the medium and the NACs are assumed to be 1ms, the clock drifts for the server and its device to be 0.99 and 0.95 correspondingly. Other parameters are also given fixed values such as *deltaT=10ms*, *relD=10ms*, *BUFLEN=5* (elements), *prec=10* (hence the digit precision is 1), *clkLimit=100*. Because of the small precision, at the beginning, the clock bound is the same in all environments. That is, *svrb=dvb=mdmb=nb=10*.

In figure 17 and figure 18, we show the task execution scheme of the system. The Sync, Follow_Up and Audio packets are delivered to the medium at time 0, 1 and 2 respectively. They are then forwarded to the NAC at time 1, 2 and 3 respectively by the medium. At time 10 a Delay_Req message is released by the device and since the NAC is currently free, the message is processed immediately and transmitted to the medium at time 11. After receiving the Delay_Req, the server sends out the Delay_Res message at time 12 and the device receives the message at time 14. This execution scheme is repeated at time 20, 40, 60, etc.

In the first cycle, since the Sync and Follow_Up messages arrived when no Delay_Req had been sent by the device, its local time is not adjusted. In the next cycle, when the Sync and Follow_Up messages arrive once again, the offset can now be computed as all parameters needed for clock synchronization are available. In this special case, the master-to-slave and slave-to-master delays are equal to 2, thereby zeroing the offset and letting the device clock remain unchanged.

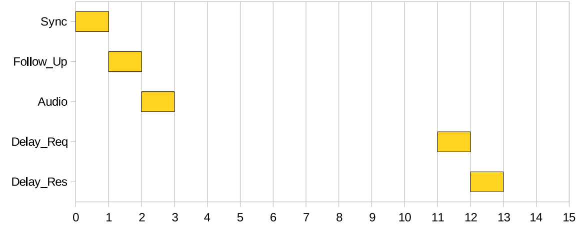


Figure 17: Task execution scheme of the medium(test1)

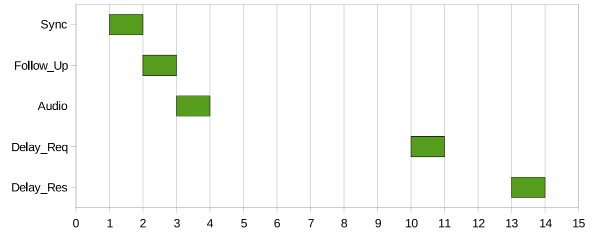


Figure 18: Task execution scheme of the NAC(test 1)

The first audio packet arrived at the device at time 4 which perfectly respects its deadline at time 10. The next audio packets would arrive at time 44, 84, etc. which will also respect their deadline of 50, 90, etc. In addition, the sizes of the audio buffer, the NAC input buffer and output buffer never exceed 1. Thus, all the schedulability properties are verified and the schedulability of the system can be guaranteed.

4.2 Test case 2

In this test case, the Audio packets are generated more frequently and its period is 5ms while the Sync interval and the Delay_Req period are 20ms and 40ms respectively. All these packets have offset 0 and the audio relative deadline is 15. Moreover, the medium latency (which is 2ms) is a little longer than the NAC latency (which is 1ms). Other parameters take the same values as in the previous test case.

Before bumping into the Error state of the $DV_AudRecvr$ timed automaton, two clock adjustments had happened as shown in table 1.

At time 0, the clock bound is the same in all environments, namely *svrb=dvb=mdmb=nb=10* due to the 1-digit-precision as in the previous case. The Sync is sent at time 0 by the server and received at time 3 by the device. Since the slave-to-master delay is not available yet, the first clock adjustment does not actually happen.

	1st	2nd	3rd
$svrb$	10	10	10
dvb	10	10	5
$mdmb$	10	10	10
nb	10	10	10
t_1	0	20	na
t_2	3	23	na
t_3	0	0	40
t_4	0	6	35
$off = t_2 + t_3 - t_1 - t_4$	0	-3	-1

Table 1: The clock adjustments happen before the automaton goes to the Error state

Audio sending	0	5	10	15	20	25	30	35
Audio receiving	11	13	15	18	27	34	44	56
Time to play	15	20	25	30	35	40	45	50
Deadline violation	No	No	No	No	No	No	No	**Yes**

Table 2: Deadline violation of audio packets

It is noticeable that the Delay_Req was released at time 0 and got out of the NAC at time 1, but because of its low priority compared to the Sync and Follow_Up messages, it has to wait until the higher priority messages pass through the medium. Thereby adding a delay to its arrival at the server which is finally 6.

At time 20, the Sync is released once again and since it has the highest priority, it arrives at the device at time 23. Now that the master-to-slave and slave-to-master delays are available, we can perform the offset calculation. In fact, the device clock is 1.5 ms behind. However, since UPPAAL does not support the real type, the offset is integerize advancing the device clock from 25 to 26 and changing the device bound from 10 to 5 as a result. After that, an audio-deadline violation takes place at time 56 before the third adjustment has happened.

Table 2 summarizes the times at which the audio packets are released, received and played. It is important to notice that the device clock has been adjusted at time 25 and elapsing two times faster than the server clock since then. And before it could be adjusted to elapse at the slower speed, it has caused one audio packet to violate its deadline.

4.3 Test case 3

In this test case, the medium latency (which is 2ms) is a much shorter than the NAC latency (which is 6ms). Other parameters take the same values as in the previous test case.

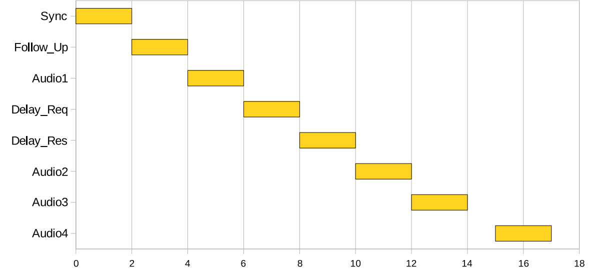


Figure 19: Task execution scheme of the medium(test3)

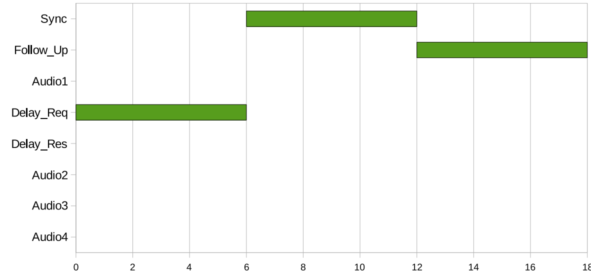


Figure 20: Task execution scheme of the NAC (test 3)

In figure 19 and figure 20 we show the task execution scheme of the system. The Sync, Follow_Up and the first Audio are delivered to the medium at time 0, 2 and 4 respectively. Also at time 0, a Delay_Req message is released by the device and since the NAC is currently free, the message is processed immediately. At time 6, the Delay_Req enters the medium while the Sync enters the NAC. After receiving the Delay_Req at time 8, the server sends out the Delay_Res message immediately. At time 10 and 12, two other Audio packets which were released at time 5 and 10 are transmitted by the medium to the NAC. Also at time 12, the NAC finishes processing the Sync and starts processing the Follow_Up. At time 15, another Audio packet is released and it arrives at the NAC at time 17 by which the NAC input buffer has overflowed because it is currently contains one Follow_Up, one Delay_Res and three other Audios. As a result, the system will soon encounter the deadlock state because when the medium has other Audio packets to forward to the NAC, it could not synchronize with the NAC input buffer automaton as the automaton is currently staying in the Error state.

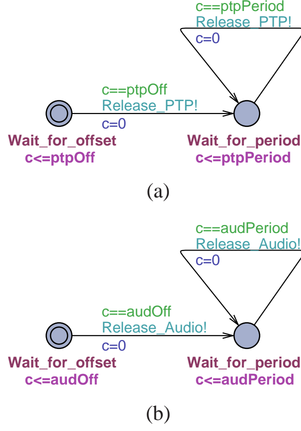


Figure 21: PTP and audio task activation automata

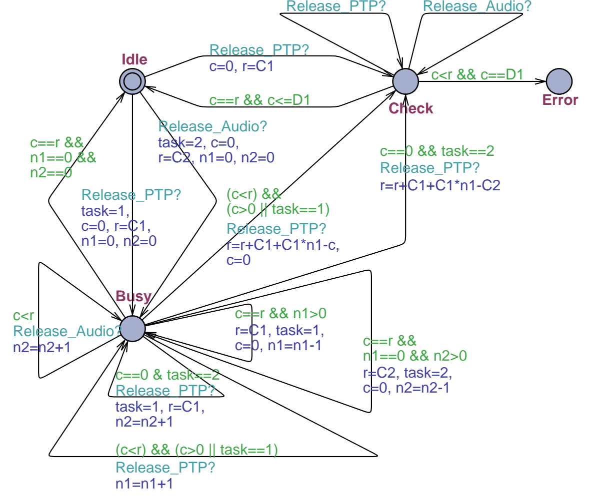


Figure 22: Schedulability checker for task PTP

5 Parametric Analysis

The UPPAAL timed automata in section 3 are very specific models which follow closely the operations of the HCS system (including the server, the medium, the NACs, the devices and the PTP protocol) given in section 2. If the details of those operations are abstracted from the system models, we will obtain abstract models describing the system at a higher level. For instance, the abstract model for the devices would be a periodic task set consisting of the PTP task and the audio receiver task. The PTP task has a higher priority than the audio receiver task as specified in section 2. In addition, the execution time of the former accounts for the total PTP load that the devices could bear and that of the latter accounts for the total delay of traversing through the medium and the NACs of the audio packets.

With all the UPPAAL models developed previously, it is now easier for the system designer to verify if there is any deadline miss with respect to the audio packets, given a fixed set of parameters. It could even be more helpful to the system designer if he could be provided with the parametric analysis on the abstract models of the system. These models gain the advantage of small complexity in applying the PTA tool over the specific models. In this section, the parametric analysis is carried out only on the abstract models of the devices to identify the parameter space that can guarantee to respect the deadline of audio packets and PTP packets. The analysis on the abstract models of the server could be done similarly.

5.1 Abstract Models

The activation automata for the PTP and audio receiver tasks are shown in figure 21. The offset, period and deadline are fixed for each task.

Based on the checkers used in [2], the schedulability checkers for the two tasks are modified to model a non-preemptive scheduling environment. That is, the audio packets can not be preempted by the PTP packets if they are currently being transmitted by the network or processed by the NACs. The preemption can only happen when the audio packets have just been released and not transmitted yet. So, when many packets are ready at the same time, the PTP would go first and the audio would back off.

The scheduler checker for the PTP task is shown in figure 22. In this figure, *D1* is the deadline of the PTP task which is less than or equal to the PTP period, *C1* and *C2* are the execution time of the PTP and audio tasks respectively.

This checker differs from the checker in [2] in the following details.

- Firstly, three additional variables are introduced. The variable *task* denotes the currently-executed task, *n1* and *n2* record respectively the number of PTP and audio packets released during the current execution.
- Secondly, at location *Busy*, when one task is being executed ($c>0$), other task instances will be queued in the respective queues. The preemption

can only take place in the situation where the audio task is about to be executed (*c==0 && task==2*) when the PTP task is ready for execution.

- Thirdly, the additional self-loops at location *Busy* are taken when the current execution is completed. If the PTP queue is not empty (*n1>0*), a PTP instance will be removed from the queue and then executed. Otherwise, an audio instance will be scheduled as long as *n2>0*. If both queues are empty, the transition from *Busy* to *Idle* is taken, indicating no tasks are ready to be executed.

- Lastly, transitions entering *Check* from *Busy* are taken when a PTP instance is (non-deterministically) chosen for checking. Again, the preemption can happen if the transition is taken at *c==0* and the task about to be executed is the audio task. Moreover, before any deadline verification, the execution time of all other PTP instances in the queue must be taken into account as they would be scheduled before the current PTP instance, that is *r* should be updated to *(r+C1*n1-c)* or *(r+C1*n1-C2)*.

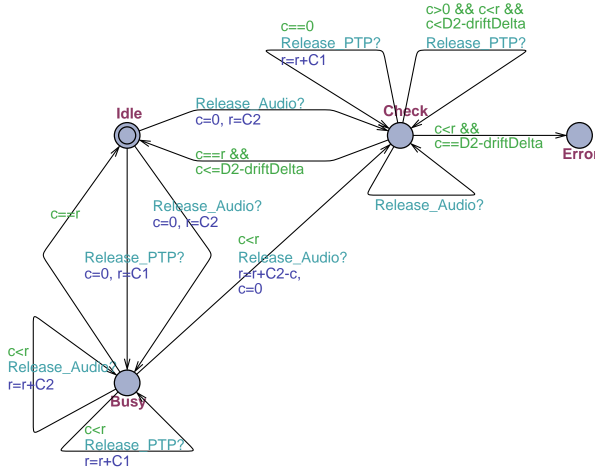


Figure 23: Schedulability checker for task Audio Receiver (hard deadline)

The scheduler checker for the audio receiver task is shown in figure 23. Similarly, the *Release_PTP* transitions are added to the *Check* state to ensure that when an audio transmission is going on (*c>0*), the PTP would back off and if that transmission is about to happen (*c==0*), the PTP can preempt it (*r=r+C1*). In the figure, *D2* is the relative deadline of audio packets. Besides, the parameter *driftDelta* is introduced to account for the offset time of the local clock compared to the server clock. The worst case happens when the local clock is substantially slower than the server clock and thus when an audio packet is received, the actual deadline to be verified would be *D2-driftDelta* instead of *D2*.

In fact, the requirement of no deadline miss is difficult to obtain in real-time environments. Therefore, in order to make the analysis more practical, the requirement can be relaxed by allowing an audio packet to sometimes miss its deadline. However, there should be no other deadline miss after one is made. In other words, the situation of two successive deadline misses should never happen. The checker adapted for this new requirement is shown in figure 24.

In the figure, three new variables are introduced. One is the boolean variable *dm* used to capture the fact that one deadline miss has already happened (*dm=true*), the others are the real variable *r1* and *r2* used to record respectively the total execution time of all PTP instances released between two consecutive audio instances and of the latter audio instance, if it is released before a deadline miss. This checker is also different from the

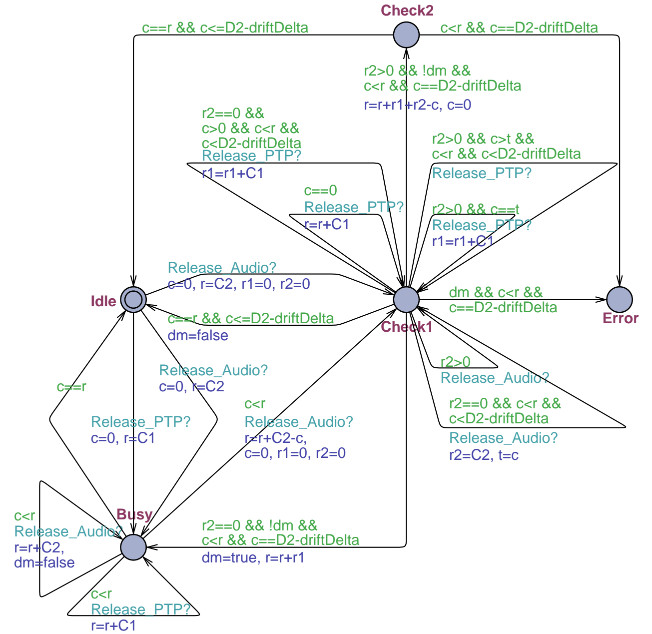


Figure 24: Schedulability checker for task Audio Receiver (soft deadline)

checker in [2] due to the above variables, the additional

	Experiment 1	Experiment 2
$ptpOff$	0	5
$ptpPeriod$	40	40
$D1$	10	10
$audOff$	0	0
$audPeriod$	10	10
$D2$	10	10

Table 3: Fixed parameter values in two experiments

$C1$
$C2$
$driftDelta$

Table 4: Free parameters of the system

	Experiment 1	Experiment 2
$Checker_{PTP}$	62	90
$Checker_{Audio}$	2	15

Table 5: Running time in minutes in two experiments

transitions from *Check1* to other locations and the introduction of the new location *Check2*.

Transitions entering *Check1* from *Idle* or *Busy* are taken when an audio instance is (non-deterministically) chosen for checking. And if another PTP instance is also ready before this audio instance is executed *(c==0)*, it would be preempted (*r=r+C1*). Then the execution time of all other PTP instances would be accumulated until another audio instance is released or it misses its deadline.

When the current audio instance finally violates its deadline:

- If one deadline miss had happened before (*dm=true*), the location *Error* is reached because of two successive deadline misses.
- Otherwise, if another audio instance has already been released (*r2>0*), the transition from *Check1* to *Check2* is taken in order to verify if it would miss its deadline the second time.
- If there was no deadline miss (*dm=false*) and no other audio instance is released before the current deadline miss, the variable *dm* is updated to *true* and the transition from *Check1* to *Busy* is taken to tolerate the first deadline miss.

5.2 Experiments

In this section, we report on the results of experimenting the above periodic task set on the PTA implemented in [2]. The following information is used as the initial constraints:

C1 > 0
C2 > 0
C1 <= D1
C2 <= D2
driftDelta >= 0

In both experiments, we use **bounded model checking** with the bound of **60** to find the feasibility region for the system. Table 3 shows the values of all fixed parameters in two experiments while the free parameters are specified in table 4. Moreover, the running time results of the two experiments are summarized in table 5. The computer used in the experiments has 1GB RAM and Intel Core 2 Duo CPU T7500 2.20 GHz. It is noticeable that the PTP checker generally runs much slower than the Audio checker which may be because the path leading to the error state of the former is much longer than that of the latter. The running time also depends on the bound used to model check the system. Using a large bound can help to find more traces to the error state, hence the feasibility region is more correct. However, the larger the bound is, the longer the running time is. So in finding the schedulable region, one must trade off between a large bound and short computation time.

Experiment 1:

For this experiment, the feasibility region in which task PTP is guaranteed to never miss its deadline is expressed in the constraints below:

1: $![(C1 = 10) \wedge (15/2 < C2 <= 10)]\wedge$
2: $![(2*C1+4*C2 > 50) \wedge (C1 <= 10) \wedge (C2 <= 10)]\wedge$
3: $![(2*C1+4*C2 = 50) \wedge (0 < C1 < 10) \wedge (15/2 < C2 <= 10)]\wedge$
4: $![(4*C1+8*C2 > 90) \wedge (2*C1+4*C2 <= 50) \wedge (C1+4*C2 > 40) \wedge (C1 <= 10) \wedge (C2 <= 10)]\wedge$
5: $![(6*C1+12*C2 > 130) \wedge (3*C1+8*C2 <= 90) \wedge (C1+4*C2 > 40) \wedge (C1 <= 10) \wedge (C2 <= 10)]$

Figure 25 graphically shows the error region for each constraint. The feasibility region of task PTP is the square with a side length of 10 excluding the total error region, as shown in figure 26.

By joining this region together with the schedulability region of task Audio expressed in the following con-

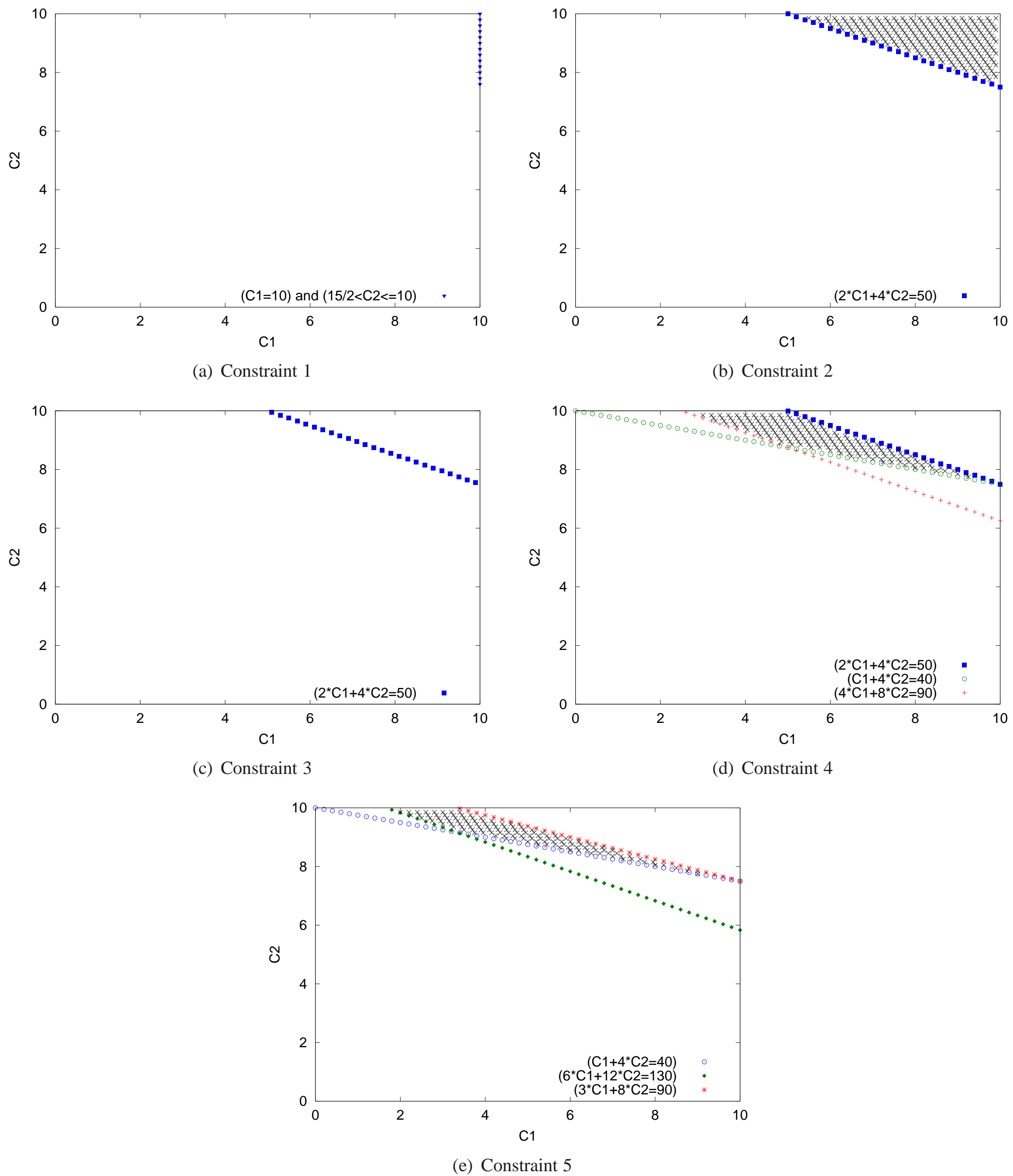


Figure 25: The error region for PTP constraints (experiment 1)

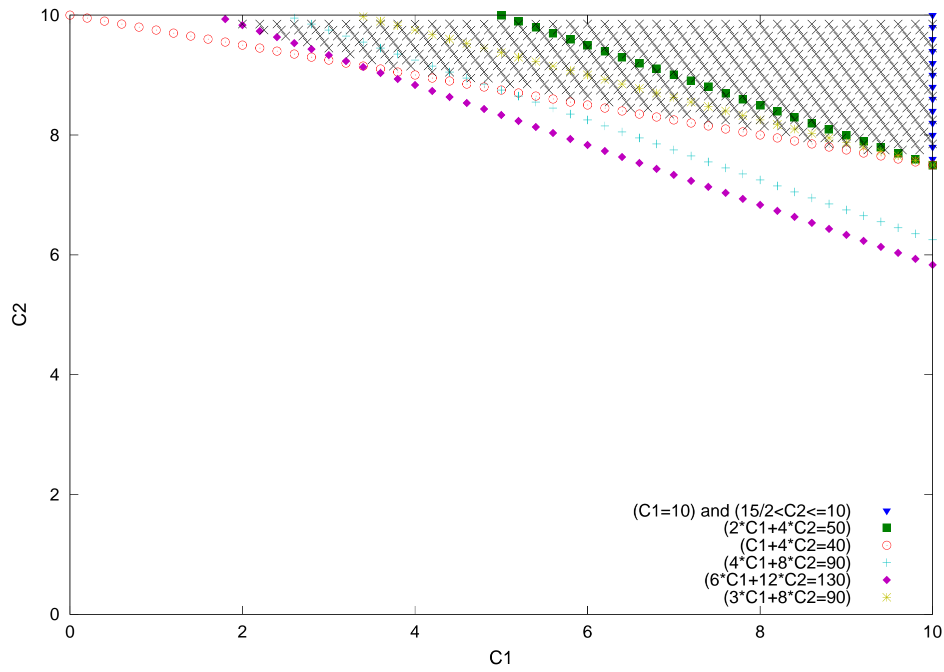


Figure 26: The total error region for task PTP (experiment 1)

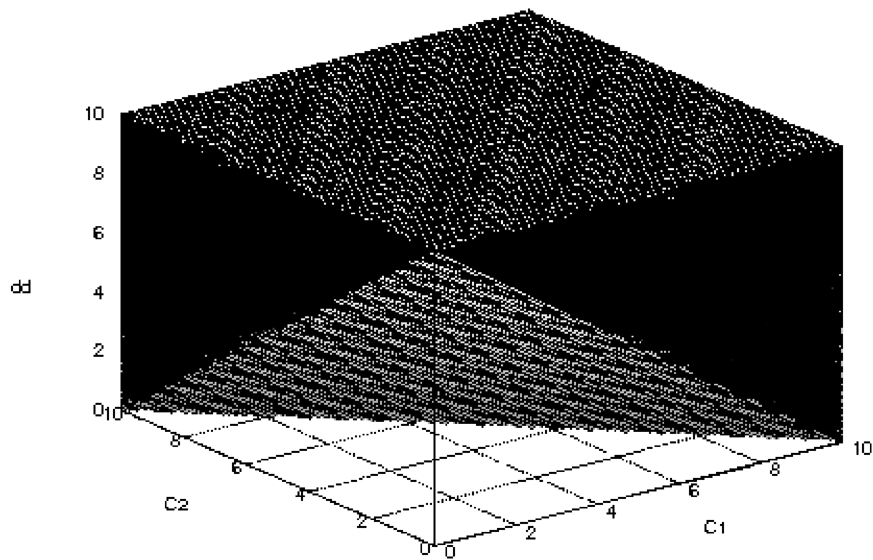


Figure 27: The total error region for task Audio (experiment 1) (*dd* stands for *driftDelta*)

straints, we would obtain the final region in which the whole system can work properly:

1: $![(dd = 10) \wedge (C1 <= 10) \wedge (0 < C2 <= 10)] \wedge$
2: $![(C1 + C2 + driftDelta > 10) \wedge (0 < C1 <= 10) \wedge (C2 <= 10) \wedge (driftDelta >= 0)]$

Figure 27 shows the error region for task Audio which has a volume of 5/6 of that of the cube with a side length of 10. The remaining volume of the cube is the feasibility region for task Audio which is a tetrahedron as figure 27.

For example, when *driftDelta = 0*, the feasibility region for the whole system is half of the base area of the cube which is the right triangle area with a cathetus length of 10. This can be easily verified by looking at the behaviour of the system. The first PTP released at time 0 does not miss its deadline because *C2<=10*. Although the first Audio instance is preempted at time 0 by the first PTP instance, it also did not violate its deadline since *C1+C2<=10*. The other three Audio instances released at time 10, 20 and 30 are not preempted as PTP instances are only released after 40 time unit. At time 40, the task arrival pattern is repeated with one PTP instance and one Audio instance released simultaneously at time 40, then three other Audio instances arrive at time 50, 60 and 70. Thus, when there is no clock drift, the system is guaranteed to be schedulable as long as *C1+C2<=10*.

When *driftDelta = 1*, the feasibility region for the whole system returned by the PTA tool is bounded by the line *C1+C2<=9*. However, there are points that should be in the feasibility region but got excluded by the tool finally. For example, with *C1 = 5* and *C2 = 5*, the first Audio instance misses its deadline but the second does not which obeys the soft deadline. And the other two Audio instances at time 20 and 30 also do not miss their deadlines. Similarly for *(C1 = 4, C2 = 6)* or for *(C1 = 3, C2 = 7)* or any other pair of values for *(C1,C2)* that satisfy the constraint *C1+C2=10*. It is noticeable that these points are included in the feasibility region for task PTP but not for task Audio. So the result returned by the tool for task Audio seems conservative in this case which needs to be investigated to find a better solution.

Experiment 2:

For this experiment, similar to what was done in experiment 1, the feasibility region in which task PTP is schedulable is expressed in the following constraints:

1: $![(C1 + C2 > 15) \wedge (C1 <= 10) \wedge (5 < C2 <= 10)] \wedge$
2: $![(5/3 < C1 <= 5) \wedge (C2 = 10)] \wedge$
3: $![(40 < C1 + 4 * C2 <= 45) \wedge (3 * C1 + 5 * C2 > 55) \wedge (C1 <= 10) \wedge (5 < C2 <= 10)] \wedge$
4: $![(5 * C1 + 9 * C2 > 95) \wedge (2 * C1 + 8 * C2 <= 85) \wedge (C1 + 5 * C2 = 50) \wedge (C1 + 4 * C2 > 40) \wedge (0 < C1 <= 10) \wedge (5 < C2 <= 10)] \wedge$
5: $![(1 < C1 <= 5/2) \wedge (C2 = 10)] \wedge$
6: $![(5 * C1 + 9 * C2 > 95) \wedge (2 * C1 + 8 * C2 <= 85) \wedge (C1 + 5 * C2 <= 50) \wedge (C1 + 4 * C2 > 40) \wedge (C1 <= 10) \wedge (5 < C2 <= 10)] \wedge$
7: $![(5 * C1 + 9 * C2 > 95) \wedge (2 * C1 + 9 * C2 > 85) \wedge (2 * C1 + 8 * C2 <= 85) \wedge (2 * C1 + 5 * C2 <= 60) \wedge (C1 + 5 * C2 > 50) \wedge (C1 <= 10) \wedge (5 < C2 <= 10)]$

And that of task Audio is expressed as follows:

1: $![(C1 <= 10) \wedge (0 < C2 <= 10) \wedge (driftDelta = 10)] \wedge$
2: $![(C2 + driftDelta > 10) \wedge (C1 <= 10) \wedge (C2 <= 10) \wedge (driftDelta >= 5)] \wedge$
3: $![(C2 + driftDelta > 10) \wedge (C1 <= 10) \wedge (5 < C2 <= 10) \wedge (0 <= driftDelta < 5)] \wedge$
4: $![(C1 + 2 * C2 + driftDelta > 20) \wedge (C1 + C2 > 10) \wedge (C1 <= 10) \wedge (5 < C2 <= 10) \wedge (driftDelta >= 0)] \wedge$
5: $![(C1 + driftDelta > 10) \wedge (5 < C1 <= 10) \wedge (0 <= driftDelta <= 5)] \wedge$
6: $![(C1 + C2 + driftDelta > 15) \wedge (C2 + driftDelta <= 10) \wedge (5 < C1 <= 10) \wedge (C2 <= 5) \wedge (driftDelta >= 0)]$

In this experiment, with a nonzero offset *ptpOff = 5*, the result becomes much more complicated because now the first PTP instance will have to experience some delay as it arrives after the first Audio instance. Thus, the trace leading to the error state will be more complicated and not as simple as in experiment 1.

6 Conclusions

In this report, the application of the PTA tool in [2] is studied by applying the tool to a distributed Heterogeneous Communication System(HCS). The reports starts with describing the system and its requirements. Next, a complete set of UPPAAL models that we have built

for the system are explained fully and clearly. These models are then validated by the ground verifications. Finally, part of the system models are converted into parametric timed automata which are run to produce the schedulability regions.

In the future, we plan to extend the models to depict fully the system, such as modeling the PTP protocol in the NAC, ensuring that the audio data is played back at end devices synchronously with a given maximal jitter (e.g. 0.1ms), etc. The parametric timed automata would then be designed in order to capture the new requirements.

7 Acknowledgments

The authors would like to thank Marius Bozga for help with modeling the system, and EADS for providing the case study.

References

[1] IST STREP 215543 COMBEST, Case Study Description and Requirements.

[2] A. Cimatti and L. Palopoli and Y. Ramadian, Symbolic Computation of Schedulability Regions Using Parametric Timed Automata, Real-Time Systems Symposium, Nov.30 2008-Dec.3 2008.

[3] H. Bowman, G. Faconti and M. Massink. Specification and Verification of Media Constraints using UPPAAL. 5th Eurographics Workshop on the Design, Specification and Verification of Interactive Systems, DSV-IS 98, Springer Verlag, 1998.

[4] K. G. Larsen, P. Patterson, and Y. Wang. UPPAAL in a nutshell. Springer International Journal of Software Tools for Technology Transfer, 1, 1997.

[5] A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE Standard 1588-2002, November 2002.

[6] R. Alur and D. L. Dill. A theory of timed automata. Theor. Comput. Sci., 126(2):183235, 1994.

[7] D. Zhang and R. Cleaveland. Fast on-the-fly parametric real-time model checking. In RTSS05, Washington, DC, USA, 2005. IEEE Computer Society.

[8] T. Hune, J. Romijn, M. Stoelinga, and F. W. Vaandrager. Linear parametric model checking of timed automata. In TACAS 01, Springer-Verlag, 2001.

[9] Etienne Andre, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. An inverse method for parametric timed automata. Electronic Notes in Theoretical Computer Science 223 (2008).