Timing Verification of an Aerial Video Tracking System Using UPPAAL

Lijun Shan
College of Computer Science,
Northwestern Polytechnical University,
Xi'an, China
lijunshancn@yahoo.com

Susanne Graf
Verimag/CNRS,
2 avenue de Vignate,
38610 Gières, France
susanne.graf@imag.fr

***Abstract*—This paper presents two different approaches for verifying the timing properties of the industrial use cases proposed as two challenges by WATERS'2015. The system under study is an aerial video system which contains two parts, a multiprocessor system and a uni-processor multitasking system. A timed automata model is constructed for each subsystem with the model checker UPPAAL. The symbolic model checking and statistical model checking functions of UPPAAL are applied to verify the models. Each of the models is modular, reusable and extensible, and can act as a general modeling framework for analyzing a type of systems.**

Keywords—Real-time systems; verification; model checking

I. INTRODUCTION

Real-time systems are widely applied in critical areas such as aerospace and aviation. The designers of a real-time system have to assure that the system can satisfy its real-time requirements before it is deployed. However, to verify such systems' timing properties is difficult due to the randomness in the behavior of such systems and their operation environment. A recent trend is to apply formal methods, especially model checking which features fully automated tools, to conduct timing verification.

This paper presents our approaches to the timing verification of two types of real-time systems in an industrial case. The system under study, proposed as a challenge by WATERS'2015, is an aerial video system which detects and tracks a moving object. The system comprises two subsystems: a multiprocessor system for video frame processing (called Video subsystem in the sequel), and a uni-processor multitasking system for tracking and camera control (called Tracking subsystem in the sequel). The latencies of the two subsystems are to be computed, respectively.

We construct a timed automata model for each subsystem, and apply the symbolic model checking and statistical model checking functions of UPPAAL [1, 2] to compute the desired values. The model for the Video subsystem captures a multiprocessor system which processes an infinite data flow. The model for the Tracking subsystem describes a multitasking system running on a real-time operating system. Each of the models is modular, reusable and extensible, and can act as a general modeling framework for analyzing a type of systems.

II. CHALLENGE 1: THE VIDEO SUBSYSTEM

A. The Video Subsystem

The Video subsystem comprises four tasks T1~T4, which process the frame flow produced by a camera and outputs the frames to a display. The timing behavior and functions of the tasks are summarized in Table 1. The time unit is *microsecond* throughout this section except given otherwise.

TABLE 1 TASKS IN THE VIDEO SUBSYSTEM

Task	Period / Trigger	Input resource	Output dest.	Exec. time	Func-tion
T1	Frame arrival	Camera	T2	28,000	Pre-process
T2	Frame arrival	T1	Register	17,000 ~ 19,000	Process
T3	13,333 +/-7	Register	Buffer	8,000	Filter
T4	40,000 +/- 4	Buffer	Display	1,000 or 10,000	D/A convert

Question 1: To compute the min/max latency for a given frame from the camera output to the display input, for a buffer size n = 1 or 3.

Question 2: To compute the minimum time distance between two frames produced by the camera that will not reach the display, for a buffer size n = 1 or 3.

B. Model of the Video Subsystem

The model comprises two parts: A system-description part consists of seven automata: PeriodGen, Camera, T1, T2, T3, Buffer and T4. A verification-supplement part consists of two automata LatencyBuffer and Monitor, which record the frames' latencies and lost frames, respectively.

1) Latency Buffer

We define an array *flowLatency*[] of clock variables as a FIFO (First-In-First-Out) buffer to track the latency of each frame under processing, and an array *latencyTiming*[] of boolean variables as the stopwatches to control the clocks in *flowLatency*[]. A stopwatch is the derivative of a clock. Setting a stopwatch to 1 (or 0) starts (or stops) the corresponding clock. Since a frame may be discarded by the

buffer hence cannot reach the display, such frames should be excluded when calculating the min/max latency. We define another clock-stopwatch pair of arrays, *reachLatency*[] and *reachTiming*[], to record the latency of those frames that finally reaches the display. The relationship between the clocks and stopwatches is specified in the following formula:

```
forall(i: frameIndex_t) flowLatency[i]'== latencyTiming[i] &&
forall(i: frameIndex_t) reachLatency[i]'== reachTiming[i]
```

The length of the four arrays, denoted by an integer constant *WindowSize*, should be the maximum number of frames under processing at the same time. Whether the value of *WindowSize* is properly set will be confirmed in the verification of the model.

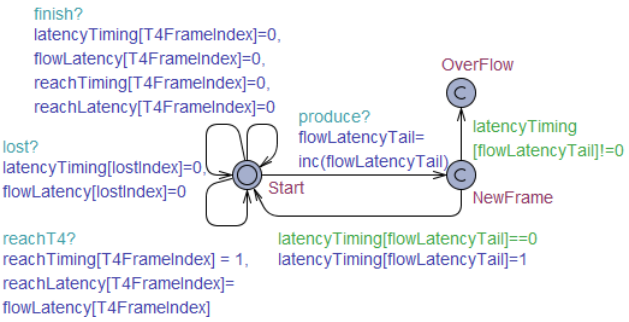


Figure 1 LatencyBuffer

The two pairs of clock-stopwatch arrays are maintained by the automaton LatencyBuffer shown in Figure 1, and manipulated in the following four occasions:

(A) Frame produced: When a frame is produced by the camera, it is associated with the clock at the tail of *flowLatency*[]. Then the frame can be referred to by the clock's index in *flowLatency*[].

(B) Frame reaches T4: A frame cannot arrive the display if and only if it is discarded by the buffer. When a frame is retrieved from the buffer by T4, which means it can finally arrive at the display, a clock in *reachLatency*[] is started.

(C) Frame output: On receiving a message *finish*?, which indicates that T4 finishes its processing of a frame, the clocks for the frame are cleared and stopped.

(D) Frame lost: On receiving a message *lost*?, which means that a frame-loss is detected, the associated clock in *flowLatency*[] is cleared and stopped.

2) Period Generator

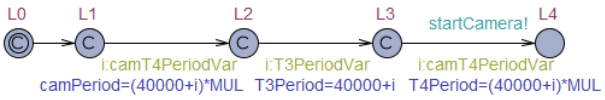


Figure 2 PeriodGen

The periods of the camera, T3 and T4 are constant throughout the running of the system, but their specific values are only known to be within their respective ranges. At the beginning of the model's execution, the automaton PeriodGen in Figure 2 chooses the periods. Note that all the periods are multiplied by a factor MUL = 3 so that T3' period can be expressed as an integer. We assume that all the tasks' offset is 0, i.e. all tasks are started simultaneously.

3) Camera

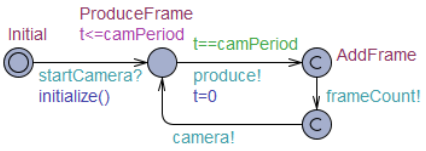


Figure 3 Camera

The automaton Camera in Figure 3 simulates the camera' periodical producing of frames. The automaton also notifies the frame's producing to the automata T1, LatencyBuffer and Monitor through synchronization messages.

4) T1 and T2

The two automata in Figure 4 simulate the two tasks T1 and T2, respectively. Each of them is triggered by the arrival of a frame, and outputs the frame after processing it. T2 sends its output to a 1-size register. A variable *registerFrameIndex* represents the index of the frame in the register .

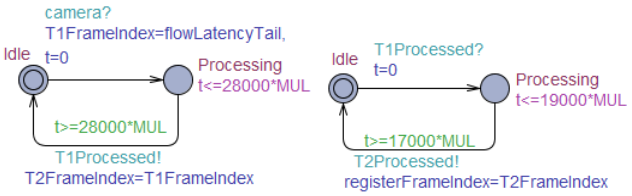


Figure 4 (A) T1 (B) T2

5) Task T3

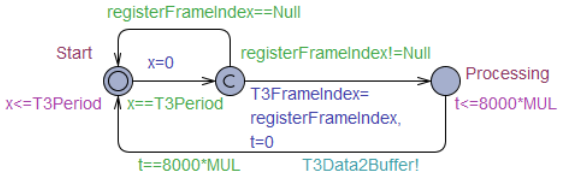


Figure 5 T3

The automaton in Figure 5 models the task T3, which reads the register with a period of 40/3 ms. If the register is empty, T3 finishes immediately. Otherwise, T3 processes the frame in 8 ms and sends it to a buffer.

6) Buffer

A FIFO buffer receives the output of T3, and is destructively read by the task T4. The buffer is implemented as an array *bufferContent*[] plus an automaton Buffer. *bufferContent*[] records the indexes of the frames in the buffer. We define a C function *add*() to write a new index to the tail of *bufferContent*[], and *pull*() to retrieve the head. The automaton Buffer tells whether an incoming frame can be accepted:

a) If the frame is new but the buffer is full, the frame is discarded.

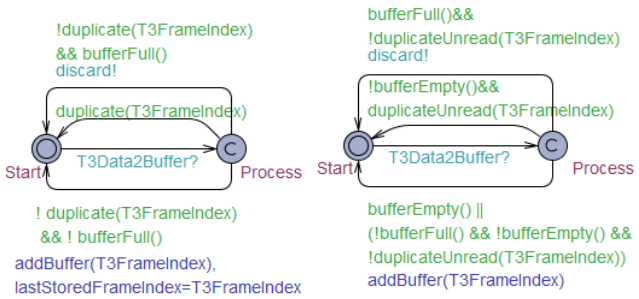
b) If the frame is duplicate, it is ignored.

c) If the frame is new and the buffer is not full, the frame enters the buffer.

But what does duplicate mean? The description of the buffer[1] in Challenge 1 can be interpreted it in two ways: a incoming frame with ID *N* is regarded as duplicate, if:

(I) A frame with ID *N* has been in the buffer, no matter whether it has been read or not.

(II) A frame with ID *N* is in the buffer.



(A) Smart Buffer (B) Normal Buffer

Figure 6 Buffer

The automaton of Figure 6 (A) implements (I), where the function *duplicate*() tells whether the incoming frame's index is the same as the last stored frame. This buffer is called Smart Buffer because it maintains a history of the last frame, no matter whether the frame has left or not. After receiving a message *T3Data2Buffer*?, the automaton takes one of the three transitions from Process to Start, which represents the three cases (a)~(c), respectively. The automaton Normal Buffer shown in Figure 6 (B) implements (II), where the function *duplicateUnread*() tells whether the incoming frame is the same as the latest frame in the buffer.

The difference between the two types of buffer is: at most one copy of a frame can enter the Smart Buffer, while two copies of a frame may enter the Normal Buffer as long as the second copy's arrival is after the first copy's leaving.

7) Task T4

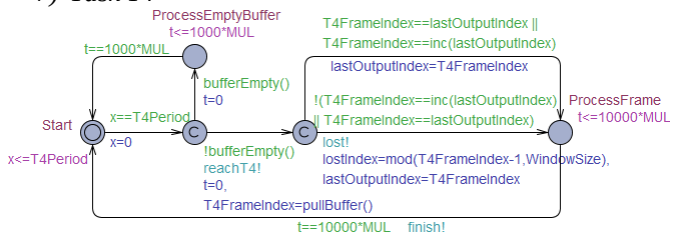


Figure 7 T4

The task T4 reads the buffer periodically. T4 spends 10,000 to process a frame, and 1,000 if there is no frame. The function *pullBuffer*() in Figure 7 destructively reads a frame from *bufferContent*[]. When T4 finishes processing of a frame, a message *finish*! is sent to the automaton LatencyBuffer.

[1] "*For each frame index value, only one single frame can be stored in the buffer. If the buffer has already stored a frame with a given index, any additional received frame with the same index is discarded.*"

In addition, the automaton T4 detects frame-losses. If the frame T4 has just read is neither same nor successive to the last one it processed, the frame between the last frame and the incoming frame is detected to be lost, and T4 sends a *lost*! message. The variable *lostIndex* records the lost frame's index.

8) Monitor

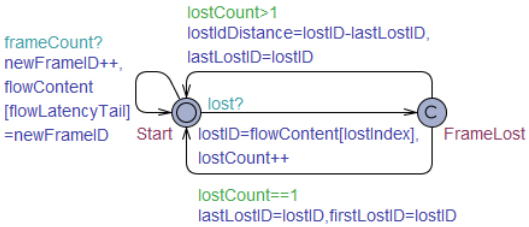


Figure 8 Monitor

The automaton Monitor shown in Figure 8 records the frame's ID, when receiving a *frameCount*? message from Camera. The number and distance of the lost frames are recorded when a *lost*? message is received.

C. Verification

For abbreviation, P_i denotes the period of the camera or task T*i* ; V_i denotes the variance scope of P_i; V_i^{Max}/V_i^{Min} denotes the max/min value in V_i; L(N) denotes the latency of the N^{th} frame; $L^{Max/Min}$ denotes the max/min latency. The model is verified with UPPAAL 4.4.18, and the hardware configuration is Intel(R) Core(TM) i7-3520M CPU 2.90 GHZ and 8GB RAM.

The frames' latency and frame loss depend on the period of the task. When $P_C = P_4 = 40$ ms, $P_3 = 40/3$ ms, the tasks run under synchronization, no frame will be lost. With the Smart Buffer, $L^{Min} = L^{Max} = 90$ ms. With the Normal Buffer, $L^{Min} = L(1) = 90$ ms, $L^{Max} = L(N) = 130$ ms, N>1.

To calculate the min/max latency and frame loss with other settings of the task periods, the following extreme cases are investigated.

1) Latency Decreases

When $P_C = V_C^{Max} = 40{,}000 + 4$, $P_4 = V_4^{Min} = 40{,}000 - 4$, each frame's latency decreases successively, hence the min latency of the system may happen. For this special case, the automaton in Figure 2 is replaced with the one in Figure 9, where Delta = 4.

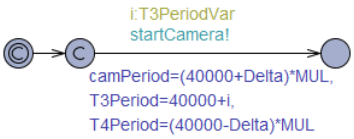


Figure 9 PeriodGen_latencyDec

The min latency is obtained by analysis or observed from the simulation plots of the query (1). The max latency is computed with the query (2), which applies statistical model checking. The statistical parameters are set by default. Note that symbolic model checking is inapplicable, because the model has an infinite state space without the tasks' synchronization. Table 2 summarizes the min/max latency and

the computation cost, including the peak (resident and virtual) memory usage and time usage for UPPAAL to compute an query.

```
simulate 1 [<=2,000,000,000] { reachLatency[i] }        (1)
E[<=2,000,000,000;  100] (max: reachLatency[i] )        (2)
```

TABLE 2 MIN/MAX LATENCY AND COMPUTATION COST ON THE QUERY (2) ($P_C = V_C^{Max}$, $P_4 = V_4^{Min}$, $P_3 \in V_3$)

Buffer type		Smart	Normal	
Buffer Size		1 or 3	1	3
Latency	L(1)	89,984	89,984	89,984
	L(2)	89,976	129,972	129,972
	L^{Min}	63	89,984	89,984
	L^{Max}	113,585	137,554	137,811
Memory	Res	7	9	10
	Virt	31	35	36
Comp. time (s)		272	320	398

```
E[<=2000000000; 100] (max: Monitor.lostCount)            (3)
E[<=2,000,000,000; 100] (min: Monitor.lostIdDistance)    (4)
E[<=2,000,000,000; 100] (max: Monitor.firstLostID)       (5)
```

Given the queries (3), (4) and (5), statistical model checking is applied to computing the number of frame losses, the distance between the lost frames' IDs and the first lost frame's ID, respectively. Table 3 summarizes the verification results and the computation cost. The verification result shows that frame loss only occurs with 1-size Normal Buffer.

TABLE 3 FRAME LOSS AND COMPUTATION COST ON THE QUERY (3) ($P_C = V_C^{Max}$, $P_4 = V_4^{Min}$, $P_3 \in V_3$)

Buffer type		Smart	Normal	
Buffer Size		1 or 3	1	3
Loss Count		0	82.95	0
1st lost frame ID		/	3403	/
Min Distance		/	3201.66	/
Memory (MB)	Resident	8	9	8
	Virtual	32	33	31
Comp. time (s)		389	372	357

2) Latency Increases

When $P_C = V_C^{Min} = 40{,}000 - 4$, $P_4 = V_4^{Max} = 40{,}000 + 4$, each frame's latency increases by 8 successively, hence the max latency of the system may happen. For this special case, we replace the automaton in Figure 2 with the one in Figure 9.

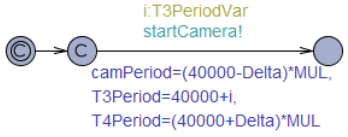


Figure 10 PeriodGen_latencyInc

The simulation plots show that the first frame has the min latency 90,016. The max latency is computed with the query (2), and the verification results are summarized in Table 4.

TABLE 4 MIN/MAX LATENCY AND COMPUTATION COST ON THE QUERY (2) ($P_C = V_C^{Min}$, $P_4 = V_4^{Max}$, $P_3 \in V_3$)

Buffer type		Smart		Normal	
Buffer Size		1	3	1	3
Latency	L^{Min}	90,016	90,016	90,016	90,016
	L(2)	90,024	90,024	130,028	130,028
	L^{Max}	135,505	212,114	139,190	214,087
Memory (MB)	Res	7	8	9	9
	Virt	31	32	33	34
Comp. time (s)		397	472	375	509

Table 5 summarizes the verification results on frame loss and the computation cost.

TABLE 5 FRAME LOSS AND COMPUTATION COST ON THE QUERY (3) ($P_C = V_C^{Min}$, $P_4 = V_4^{Max}$, $P_3 \in V_3$)

Buffer type		Smart		Normal	
Buffer Size		1	3	1	3
Loss Count		3	1	180.82	2
1st lost frame ID		5712.14	15581.5	917.62	10740.1
Min Distance		4631.96	/	301.88	4952.01
Memory (MB)	Res	8	9	9	10
	Virt	31	33	35	37
Comp. time (s)		329	502	360	497

3) Summary

TABLE 6 MIN AND MAX LATENCY IN ALL CASES

Buffer type		Smart		Normal	
Buffer Size		1	3	1	3
Latency	Min	63	63	89,984	89,984
	Max	135,505	212,114	139,190	214,087

Table 6 summarizes the max/min latency with the Smart Buffer and the Normal Buffer.

III. CHANLLENGE 2: THE TRACKING SUBSYSTEM

A. The Tracking Subsystem

TABLE 7 TASKS IN THE TRACKING SUBSYSTEM

Task	Period / Trigger	Functions
T2PR	40 +/- 0.01% (ms)	Processing
T6TC	100 (jitter=20) (ms)	Tracking control
T5TP	Called by T6	Target position prediction
T7CC	Called by T6	Camera control

The Tracking subsystem is a concurrent multitasking system which comprises three tasks: T6, T5 and T7. The three tasks are mapped to a CPU together with T2, one of the tasks in the Video subsystem. Table 7 summarizes the periods/ triggers and functions of the four tasks, where the tasks are listed by their priorities in a descending order. T2PR and T5TP

have access to a shared resource. The access to the shared resource takes 2ms for each task.

Question 1: To compute the best-case and worst-case end-to-end latencies from activation of T6 to termination of T7 for a jitter value j = 0 or 20 ms.

Question 2: To compute the optimum priority assignment minimizing the worst-case latency for a jitter value j = 0 or 20 ms.

B. Model of the Tracking Subsystem

Our approach to the timing verification of the Tracking subsystem is inspired by the schedulability analysis approach proposed by the UPPAAL team [3]. The model of the Tracking subsystem consists of an automaton for the scheduler, an automaton for the idle task, and a template for periodic tasks.

1) RTOS

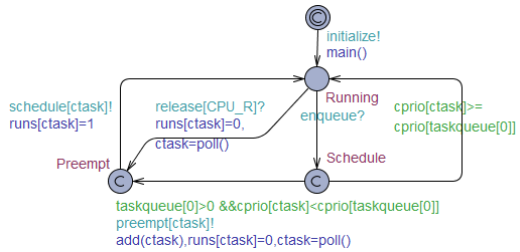


Figure 11 Scheduler

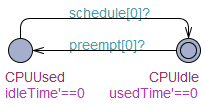


Figure 12 IdleTask

To exhibit the parallel running of the multiple tasks, the model has to describe how the RTOS schedules the tasks. The automaton for the CPU scheduler is shown in Figure 11. The function *main*() assigns initial priorities to all tasks according to their IDs. An array *taskqueue* represents the queue of ready tasks. The task queue is manipulated by the functions *poll*() and *add*(): *poll*() destructively read the head of the queue, and *add*() adds a task to the queue and sorts the queue by the tasks' priorities to a descending order. A variable *ctask* denotes the ID of the current running task.

When the task queue is empty, an idle task, whose priority is 0 (the lowest) and ID is 0, runs on the CPU. The automaton IdleTask is shown in Figure 12. The idle task runs before any task gets ready. After some time, a task gets ready, joins the ready queue and sends an *enqueue*! message to Scheduler. On receiving a *enqueue*? message, Scheduler takes the transition from the location Running to Schedule. If the running task's priority is lower than the head of the task queue, Scheduler takes the transition from Schedule to Preempt. The function *add*() adds the preempted task to the ready queue. The stopwatch *runs*[*ctask*] is set to 0, which stops the timing of the preempted task's execution. *ctask* is updated by calling the function *poll*(), which retrieves the head of the task queue.

2) Operations in Tasks

```
typedef struct {
   funtype_t cmd;
   time_t minDelay;
   time_t maxDelay;
} fun_t;
```

Figure 13 Data structure of operations

To display each task's execution, 4 types of commands are defined: COMPUTE, LOCK, UNLOCK and END. COMPUTE represents all kinds of operations that need some CPU time. LOCK and UNLOCK are used for mutual exclusive access of the shared resource. The data structure for specifying an operation is defined as a C struct *fun_t*, as shown in Figure 13, where *minDelay* and *maxDelay* represents the min and max CPU time of an operation, respectively. The delay of a LOCK or UNLOCK or END operation is 0. The operation flow of a task is an array whose elements are instances of the struct *fun_t*.

Since T5TP and T7CC are sequentially invoked by T6TC, the three tasks can be combined into one:

(1) T5TP: it is invoked by a synchronous call of T6TC, hence can be embedded into the suspension section of T6TC, whatever its priority is.

(2) T7CC: if its priority is higher than T6TC, its execution is inserted before the last COMPUTE operation of T6TC. Otherwise, it runs after the last COMPUTE operation of T6TC. Since T7CC is pure COMPUTE, its priority does not influence the timing properties of the system.

```
const Flow_t Processing = // T2
{
   { LOCK,              0,    0    },    //1. Lock shared resource
   { COMPUTE,      2000,  2000 }, //2. Write into the resource
   { UNLOCK,           0,    0    },    //3. Release shared resource
   { COMPUTE,    15000, 15000 }, //4. Compute for 15 ms
   FIN, FIN
};
const Flow_t TrackingControl =  //T567
{
    { COMPUTE,  4000,   4000  },    //1. TC: Action1
    { LOCK,        0,     0    },        //2. TP: (2.1) Lock resource
    { COMPUTE,  2000 ,  2000  },   //  (2.2) Write the resource
    { UNLOCK,    0,     0    },        //  (2.3) Release resource
    { COMPUTE,  26000,  34000 }, //  5+10+5+14
     FIN
};
```

Figure 14 The operation flows of T2 and T567

The combination of T5, T6 and T7 is called T567 in the sequel. Figure 14 shows the operation flows of T2 and T567.

3) Periodic Tasks

We build a timed automaton called PeriodicTask, as the template of all periodic tasks, to describe the state transitions of a periodic task from the RTOS' viewpoint. The parameters of the template PeriodicTask include the task's ID and operation flow. When the parameters are assigned with

concrete values, as shown in Figure 15, the template is instantiated to a timed automaton for each task.

```
//                                  taskid,          flow,
Task2PR_P1  = PeriodicTask(Task2PrID,   Processing);
Task6TC_P2 = PeriodicTask(Task6TcID,  TrackingControl);
```

Figure 15 Instantiation of periodic tasks

The template for periodic tasks is shown in Figure 16. Take T567 as an example of periodic task. After initialization, the automaton moves to the location Ready. When T567 is scheduled, the automaton goes to GotCPU, and then to different locations depending on the types of operations in the operation flow. Since the first operation in T567 is COMPUTE, the automaton takes a transition to Computing, and stays at Computing until the specified span of the operation is spent. At Computing, a stopwatch expression (*sub'*==*runs*[*id*]) imitates preemptive scheduling. When a task is preempted, the clock variable *sub* stops and the Boolean variable *runs*[*id*] is set to 0, indicating that the task stops running. After executing an operation, the automaton goes to the location Next, then the task will execute the next operation in the operation flow. The remaining operations in the operation flow are executed sequentially until reaching the end of the program. Then the automaton goes to Release, representing the task releasing the CPU, and then to Idle. On the arrival of *Period+Offset*, the automaton goes to Ready, then the task joins the task queue again.

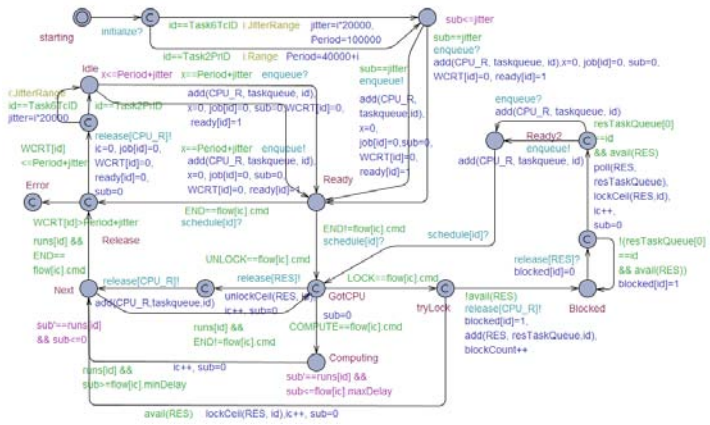


Figure 16 Template for periodic tasks

The function *lockCeil*() increases the resource owner's priority. Similarly, when a task's use of the resource finishes, *unlockCeil*() resets the task's priority to the original.

C. *Verification*

We firstly conduct an empirical analysis. Since the worst case execution time (WCET) of T567 is 40 ms and the period of T2 is 40 +/- 0.01%, T2 will run once or twice during the execution of T567. When T567 and T2 arrive at the same time, WCRT(T567) may cover two runs of T2. WCRT(T567) = CET(T2) + WCET(T567) + CET(T2) = 17*2 + 40 = **74** (ms). BCRT(T567) only covers one run of T2, hence BCRT(T567) = BCET(T567) + CET(T2) = 32 + 17 = **49** (ms).

Secondly, given the query (5), the worst case latency of T567 can be calculated with the UPPAAL statistical model checking, where 2 is the ID of T567 in the model. The result shown in Table 8 conforms to the empirical analysis result.

E[<=1000000000; 100] (max: WCRT[2]) (5)

TABLE 8 LATENCY IN DIFFERENT SITUATIONS

Jitter	Worst-case latency
0	73952
20	73998.2

The analysis and verification results show that the jitter does not affect the WCRT and BCRT. The relative priority of T5, T6 and T7 does not influence the WCRT and BCRT. Since WCET(T567) + WCET(T2) > Period_T2, the system is schedulable only if the priority of T2 is the highest.

IV. CONCLUSION

We constructed timed automata models for a multi-processor system and a uni-processor multitasking system, and applied symbolic and statistical model checking of UPPAAL to verify their timing properties. The main effort was building the two models, which took 20 and 5 man-days, respectively. The time for building the models may vary for modelers with different proficiency. The weakness of this method includes: (1) With symbolic model checking, state space explosion may happen. (2) In the statistical model checking of UPPAAL, the upper bound of time limit is $2 * 10^9$. For the Video system, this time span only represents $2 * 10^6 / 3 = 666{,}667$ (ms). (3) Statistical model checking may be time-consuming. It may take 10 ~ 30 minutes to compute a query with the max time limit.

REFERENCES

[1] G. Behrmann, A. David, K.G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks: "UPPAAL 4.0", in Editor (Ed.)^(Eds.): Book UPPAAL 4.0 (IEEE, 2006, edn.), pp. 125-126

[2] P. Bulychev, A. David, K.G. Larsen, A. Legay, M. Mikucionis, and D.B. Poulsen: "Checking and distributing statistical model checking": NASA Formal Methods (Springer, 2012), pp. 449-463

[3] A. David, K.G. Larsen, A. Legay, and M. Mikucionis: "Schedulability of Herschel revisited using statistical model checking", International Journal on Software Tools for Technology Transfer, 2014, pp. 1-13