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 a challenge by WATERS'2015. The system under study is an aerial video system which contains two parts, one multiprocessor system and one 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 timing properties of the models. Both models are 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 full automated tools, to conduct timing verification.

This paper presents our approaches to verifying two types of real-time systems through an industrial use 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: one multiprocessor system for video frame processing (called Video subsystem in the sequel), and one uni-processor multitasking system for tracking and camera control (called Tracking subsystem in the sequel). The main aim of the verification is to compute the latencies of the subsystems.

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

II. CHALLENGE ONE: THE VIDEO SUBSYSTEM

A. The Video Subsystem

The Video subsystem, which comprises four tasks T1~T4, processes the frame flow produced by a camera and outputs the frames to a display. A register and a buffer store the output of T2 and T3, respectively. 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	Out. 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 = 1or 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: system description which simulates the system's behavior, and verification solution which records the information needed for the verification. The system description part, consists of seven automata: PeriodGenerator, Camera, T1, T2, T3, Buffer and T4. The verification solution part consists of two automata: LatencyBuffer and Monitor, where the former tracks the frames' latencies and the latter records the lost frames.

1) Latency Buffer

A FIFO (First-In-First-Out) buffer, implemented as an array *flowLatency*[] of clock variables, tracks the time passed for each frame under treatment. Another array *latencyTiming*[]

with the same size acts as stopwatches to control the clocks in *flowLatency*[]. A stopwatch is a boolean variable, as the derivative of a clock. The effect of setting a stopwatch to 1 (or 0) is to start (or stop) the corresponding clock. The length of the two arrays, denoted by an integer constant *WindowSize* = *4*, is set to be the maximum number of frames in the Video system at the same time. Whether the value of *WindowSize* is properly set will be confirmed with the verification of the model.

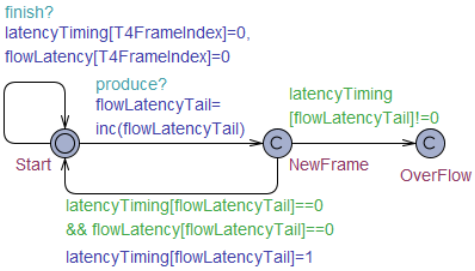


Figure 1 LatencyBuffer

The manipulations of the two arrays are controlled by the automaton LatencyBuffer, as shown in Figure 1. On receiving a message *produce*?, indicating that a new frame is produced by the camera, the transition from the location Start to NewFrame is taken. Meanwhile the variable *flowLatencyTail*, which is the pointer to the tail of *flowLatency*[] and *latencyTiming*[], moves forward. Now if the tail elements of the two arrays are both 0, which means the clock at the tail of *flowLatency*[] is idle, the transition from NewFrame to Start is taken, and the clock is started. Consequently, the latest frame is associated with the clock at the tail of *flowLatency*[]. Since each frame is assigned with a clock, the frame can be referred to by the clock's index in *flowLatency*[].

If the stopwatch at the tail of *latencyTiming*[] is not 0, it means that the corresponding clock is still occupied by a frame under processing. In this case, the transition from the location NewFrame to OverFlow is taken. The overflow of *flowLatency*[] is probably because it is too short to track all frames in the system.

On receiving a message *finish*?, indicating that T4 finishes its processing of a frame, the self-transition on Start is taken, which clears and stops the clock for the frame.

2) Period Generator

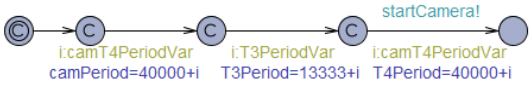


Figure 2 PeriodGen

The periods of the camera, T3 and T4 are constant throughout the infinite running of the system, but their specific values are only known to be within their respective ranges. In the model, the periods have to be chosen before the system runs. The automaton PeriodGen assigns values to the periods *camPeriod*, *T3Period* and *T4Period* within their respective ranges.

3) Camera

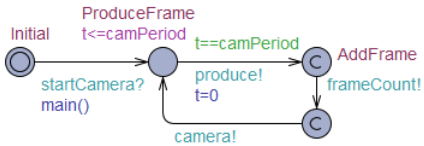


Figure 3 Camera

The automaton Camera is shown in Figure 3. At the beginning, a function *main*() is called to initialize some variables used in the model. On the arrival of a period *camPeriod*, Camera sends three synchronization messages sequentially: *produce*! to LatencyBuffer, *frameCount*! to Monitor, and *camera*! to T1. Note that *produce*! must be sent before *camera*!, because T1 uses the new frame's index, which is obtained when LatencyBuffer assigns a clock to the frame.

4) T1and T2

The automaton for task T1, as shown in Figure 4 (A), is triggered by a message *camera*?. Meanwhile, the variable *T1FrameIndex*, which represents the index of the frame under T1's processing, is updated to *flowLatencyTail*. After 28,000, T1 finishes and sends a message *T1Processed*! to the automaton T2.

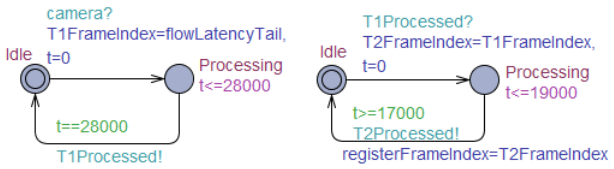


Figure 4 (A) T1 (B) T2

The behavior of the task T2 is similar to T1. As shown in Figure 4 (B), T2 is triggered by a message *T1Processed*?. Then *T2FrameIndex*, the index of the frame under T2's processing, is updated. After a span of time which ranges over [17,000, 19,000], T2 finishes processing and sends a message *T2Processed*! to the automaton Register. Meanwhile, *registerFrameIndex*, a variable representing the content of the register is updated.

5) Task T3

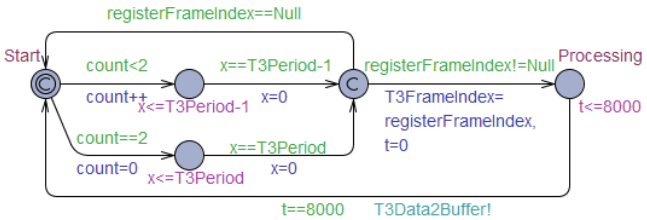


Figure 5 T3

The task T3 reads the register with a period *T3Period*. As Figure 5 shows, if the register is empty, T3 finishes immediately. Otherwise, T3 reads the frame in the register, processes it in 8,000, and sends the frame to the buffer, modeled by sending a message *T3Data2Buffer*! to the automaton Buffer. Note that the period of T3, 40/3 ms, is not equal to 13,333 or 13,334 microseconds. We use a variable *count* to simulate that the period of three runs of T3 is 40 ms, where T3Period has the value of 13,334.

6) Buffer

The buffer receives T3's output periodically, and is destructively read by T4 with a different period. The FIFO buffer is implemented as an array *bufferContent*[] plus an automaton Buffer. *bufferContent*[] records the frames existing in the buffer with their indexes in *flowLatency*[]. The function *add*() write a new item to the tail of *bufferContent*[], while *pull*() retrieves its head. The automaton Buffer tells whether a incoming frame with index *T3FrameIndex* can be accepted:

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

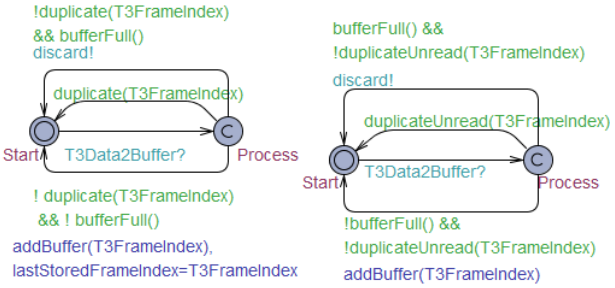
b) If the incoming frame is duplicate, it is ignored.

c) If the incoming frame is new and the buffer is not full, the frame is added into 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 to the buffer, no matter read or not.

(II) A frame with ID *N* is in the buffer and not read.



(A) Smart Buffer (B) Normal Buffer

Figure 6 Buffer

The automaton shown in Figure 6 (A) implements (I), where the function *duplicate*() tells if the incoming frame's index is same to the last stored frame. It is called Smart Buffer because the buffer maintains a history of the last frame, no matter the frame has left the buffer or not. After receiving a message *T3Data2Buffer*? and taking the transition from the location Start to Process, the automaton takes one of the three transitions from Process to Start, which represents the three cases (a)~(c), respectively. In this case, the interval of accepting frame *N* and frame (*N*+1) is the period of Camera i.e. *camPeriod*, because the duplicate copies of the frame *N* which arrive during this interval are discarded.

The automaton Normal Buffer shown in Figure 6 (B) implements (II), where the function *duplicateUnread*() tells if the incoming frame is same to the last frame existing in the buffer. In this case, two copies of a frame can enter the buffer, as long as the second copy's arrival time is later than the first copy's leaving time.

7) Task T4

[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.*"

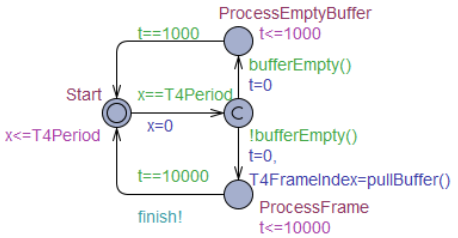


Figure 7 T4

The task T4 reads the buffer with a period *T4Period*. If the buffer is empty, T4 spends 1,000. Otherwise, it spends 10,000 for processing a frame. As shown in Figure 7, the function *pullBuffer*() destructively reads a frame from the buffer. When T4 finishes processing of a frame, the transition from the location ProcessFrame to Start is taken, which sends a message *finish*! to the automaton LatencyBuffer.

8) Monitor

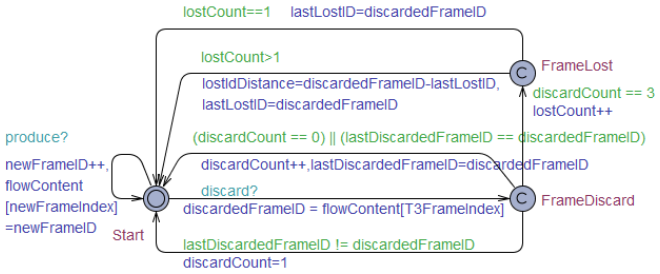


Figure 8 Monitor

The automaton Monitor shown in Figure 8 records the latest frame's ID and the distance between lost frames by synchronizing with the automaton Camera and Buffer, respectively. The self-transition at the location Start is taken when Monitor receives a *produce*? message sent by Camera. Then the variables *newFrameID* and *flowContent*[] have their values updated to record the frames's ID.

The rest part of the automaton counts the lost frames. On receiving a *discard*? message, the transition from Start to FrameDiscard is taken. The variable *discardCount* records how many copies of a frame are discarded. A frame is lost only when its three copies are all discarded. Once a frame is found lost, the transition from the location FrameDiscard to FrameLost is taken. When the number of lost frames is more than one, the variable *lostIdDistance* records the distance between two lost frames.

C. Verification of Min/Max Latency

A frame's latency starts from its generation by the camera till the finish of T4. The variance in the periods of the camera and T4 influences the frame's latency. Since the automaton PeriodGen leads to state explosion when verifying the model, we only consider typical and extreme cases. To apply symbolic model checking for computing min/max latency, Monitor is excluded from the model, so that the model contains no variables ranging over an infinite domain.

1) Standard

When both the camera and T4 have the *standard* period, i.e. camPeriod = 40,000, T4Period = 40,000, the max latency can be computed by symbolic model checking with query (1).

sup: flowLatency[0], flowLatency[1], flowLatency[2], flowLatency[3] (1)

The min latency cannot be directly computed. In any settings, the first frame has the min latency 90 ms, which can be confirmed by the simulation with query (2).

simulate 1 [<=1000000] {flowLatency[0] , flowLatency[1], flowLatency[2], flowLatency[3] } (2)

Table 2 Min/max latency in Standard case

Buffer type		Smart	Normal
Buffer Size		1 or 3	1 or 3
Latency	Min	90	90
	Max	90	130

The result is summarized in Table 2. The result can be justified by manual analysis: With Smart Buffer, each frame can enter the buffer only once. Therefore the latency of each frame is 90. However, with Normal Buffer, the first frame enters the buffer twice, because its third copy arrives at the buffer at 128 ms, which is later than the first copy's retrieval time 120 ms. Consequently, the third copy of the 1st frame enters T4 again and postpones T4's processing of later frames by one period of T4. Note that the buffer's size influences the time when the second frame entering the buffer (at 167 ms when BufferSize=1; at 141 ms when BufferSize = 3), but does not the latency.

2) Latency Increase

When the camera has the least period, and T4 has the greatest period, i.e. camPeriod = 40,000-4, T4Period = 40,000+4, after the second frame, later frames' latency increases by 8 successively. The max latency can be computed by symbolic model checking with query (1). The first frame has the min latency 90 ms, which can be confirmed by the simulation with query (2). The verification result is summarized in TABLE 3.

TABLE 3 Min/max latency (camPeriod = 40,000-4, T4Period = 40,000+4)

Buffer type		Smart	Normal
Buffer Size		1 or 3	1 or 3
Latency	Min	90,016	90,016
	2nd frame	90,024	130,028
	Max	159,984	159,984

3) Latency Decrease

If the camera has the greatest period, and T4 has the least period, i.e. camPeriod = 40,000+4, T4Period = 40,000-4, after the first frame, the latency of the frames decreases 8 microseconds by each cycle. The min latency in the case of Smart Buffer is obtained through simulation.

TABLE 4 Min/max latency (camPeriod = 40,000+4, T4Period = 40,000-4)

Buffer type		Smart	Normal	
Buffer Size		1 or 3	1	3
Latency	1st	89,984	89,984	89,984
	2nd frame	89,976	129,972	129,972
	Min	77,080	89,984	89,984

Table 2 summarizes the min/max latency in the case of Smart Buffer and Normal Buffer.

Table 5 Min/max latency in all cases

Buffer type		Smart	Normal
Buffer Size		1 or 3	1 or 3
Latency	Min	77,080	89,984
	Max	159,984	159,984

D. *Verification of Lost Frames*

Statistical model checking is applied to compute the number of lost frames on the model, which includes the Monitor automaton. TABLE 5 summarizes the verification result, which shows that the no frame is lost in any case. When using a Normal Buffer with size 1, some frames are discarded but not lost .

TABLE 6 COUNTS OF LOST FRAMES AND THEIR DISTANCES

Buffer Type	Buffer Size	Lost Count	Discard Count
Smart	**1**	0	0
	3	0	0
Normal	**1**	0	1.46
	3	0	0

III. CHANLLENGE 2: THE TRACKING SYSTEM

A. *The Tracking Subsystem*

The Tracking subsystem is a concurrent multitasking system which comprising 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 6 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.

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

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

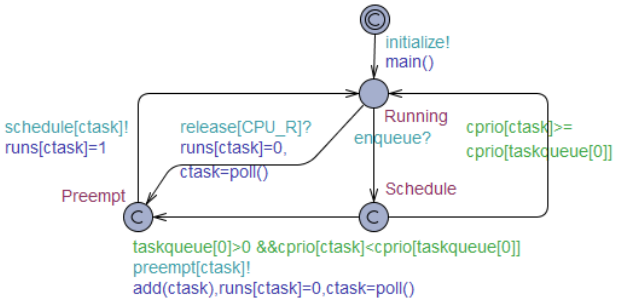


Figure 9 Scheduler

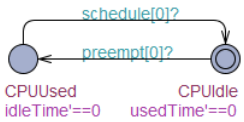


Figure 10 IdleTask

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 10. 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 11.

At the beginning, the function *main*() is called. The idle task runs before any task getting ready. At 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 currently running task's priority is higher than the head of the task queue, Scheduler simply takes the transition from Schedule back to Running. Otherwise, it takes the transition to Preempt. The function *add*() is called to add the preempted task to the ready. 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;      // type of command
   time_t delay;       // CPU time needed
} fun_t;
```

Figure 11 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 to run on CPU. 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 12, where *delay* represents the CPU time of an operation. The delay of a LOCK/UNLOCK/END operation is 0.

The operation flow of a task is an array whose elements are instances of the struct *fun_t*. Then the program of a task can be specified as an operation flow using the 4 types of operations. Note that as the time an operation spends may vary within a given span, a task's operation flow has two versions: maximum time and minimum time.

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, so it 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. As T7CC is pure COMPUTE, its priority does not influence the timing property of the system.

```
const Flow_t Processing = // (4) Period = 40 ms +/- 0.01%
{
   { 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 =
{
    { 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 12 The operation flows of T2 and T567

The combination of T5, T6 and T7 is called T567 in the sequel. Figure 13 shows the operation flows of T2 and T567.

3) Periodic tasks

We built a timed automaton called PeriodicTask, as the template of all periodic tasks, to describe the state transitions of a periodic task from the viewpoint of the RTOS. The parameters of the template PeriodicTask include the task's ID, its offset (how far into the cycle the task is released), its period, and its operation flow. When the parameters are assigned with concrete values, as shown in Figure 14, the template is instantiated to a timed automaton for each task.

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

Figure 13 Instantiation of periodic tasks

The template for periodic task is shown in Figure 15. 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, so that the task will execute the next operation in the operation flow. Likewise, 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.

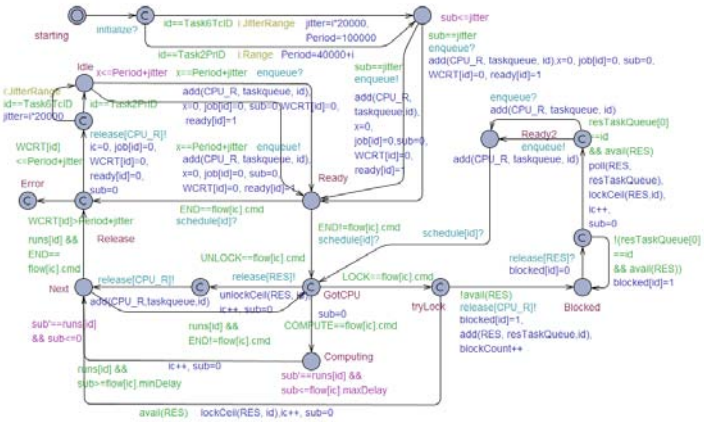


Figure 14 Template for periodic tasks

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

The analysis aims to compute the best-case and worst-case end-to-end latencies from activation of T567 to termination of T7CC for a jitter value j = 0ms and j = 20ms, respectively, and the optimum priority assignment minimizing the worst-case latency for a jitter value j = 0ms and j = 20ms.

Given the query (5), the worst case latency can be calculated, the result is shown in TABLE 7.

E[<=1000000000; 100] (max: WCRT[2]) (1)

TABLE 8 LATENCY IN DIFFERENT SITUATIONS

Jitter	Best-case latency	Worst-case latency
0	49	73952
20	49	73998.2

Since the worst case execution time (WCET) of T567 is 40, T2 will run once or twice during the execution of T567. When T567 and T2 arrive at the same time T, WCRT(T567) may cover two runs of T2. WCRT(T567) = CET(T2) + WCET(T567) + CET(T2) = 57 (ms). The best case response time (BCRT) of T567 is not directly computable. BCRT(T567) only covers one run of T2. BCRT(T567) = BCET(T567) + CET(T2) = 32 + 17 = 49 (ms).

IV. CONCLUSION

Providing a precise model of a system (at the right level of abstraction) and the use of model-checking is – when feasible – an optimal way to get deep insight about the functioning of a system. In this case study, we constructed timed automata models for two types of real-time systems, i.e. multi-processor system and uni-processor multitasking system, and used symbolic and statistical model checking of UPPAAL to verify their timing properties. With the automated model checking tool support, the main effort is building the models. The modeling of Video subsystem took 20 man-day. Based on our previous work[4], the modeling of Tracking subsystem took 5 man-days.

The weakness of this method includes: symbolic model checking may lead to state explosion; statistical model checking of UPPAAL has a time limit of 1,000,000,000.

REFERENCES

1. Behrmann, G., et al. *UPPAAL 4.0*. in *Third International Conference on Quantitative Evaluation of Systems (QEST 2006)*. 2006. IEEE.
2. Bulychev, P., et al., *Checking and distributing statistical model checking*, in *NASA Formal Methods*. 2012, Springer. p. 449-463.
3. David, A., et al., *Schedulability of Herschel revisited using statistical model checking.* International Journal on Software Tools for Technology Transfer, 2014: p. 1-13.
4. Shan, L., et al., *Formal Verification of Lunar Rover Control Software Using UPPAAL*, in *19th International Symposium on Formal Methods (FM 2014), LNCS 8442*, P.P. C. Jones, and J. Sun, Editor 2014, Springer International Publishing Switzerland: Singapore. p. 718--732.