Model Checking Task Sets with Preemption Thresholds

Mitchell L. Neilsen
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS, USA

Abstract

Several models have been developed for symbolic schedulability analysis of periodic, preemptive task sets in real-time systems [1, 11]. However, these models cannot be used to analyze task sets with preemption thresholds. In this paper, we present a new model with a two-clocks scheduler that can be used to efficiently analyze periodic task sets with preemption thresholds. The new model can also be used to compute an optimal set of preemption thresholds for a given priority assignment. In particular, it can be used to compute a feasible set of preemption thresholds that are as small as possible.

Keywords: model checking, preemption threshold, real-time system, scheduling theory, worst-case response time

1 Introduction

In classic real-time scheduling theory, tasks are frequently assumed to be periodic. To relax the traditional constraints on task arrival times, automata can be used to model task arrival patterns [1]. Such models are expressive enough to model real-time tasks that are periodic, sporadic, preemptive or non-preemptive, and tasks with additional precedence and resource constraints. Typical real-time scheduling algorithms can be easily generalized to automata. An automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable. It has been shown that the schedulability analysis problem for such models is decidable [5].

Schedulability analysis can be performed in a manner similar to response time analysis in classical real-time scheduling theory. Real-valued clocks can be used to model task execution and deadlines. The model also includes an automaton which is used to represent the scheduler. Then, a model checker, such as UPPAAL [2, 3], can be used to check for missed deadlines and calculate the worst-case response time for each task. Similar models can be used to verify the correctness of other distributed algorithms [9, 10].

Many existing real-time models have been analyzed using symbolic schedulability analysis [6, 7, 11]. However, if tasks in a periodic task set are allowed to be assigned preemption thresholds – dynamic priorities used to determine if a running task can be preempted – then these existing models cannot be used. Preemption threshold scheduling is important for real-time sensor networks and real-time embedded systems because premption thresholds can be used to enable energy and memory efficient scheduling in real-time embedded systems [4, 8]. In a previous paper, we developed a model that can be used to determine if a task sets is feasible if the preemption thresholds are previously assigned [12]. The goal of this paper is to develop new models that can be used to both analyze task sets with preemption thresholds and find an optimal set of preemption thresholds that are minimal.

The rest of the paper is organized as follows: the next section describes the input language for the model checker UPPAAL. Section 3 describes a simple two-clocks model that can be used to correctly test for the schedulability of task sets with preemption thresholds. The model can also be used to derive an optimal set of minimal preemption thresholds. Finally, Section 4 provides a brief summary.

2 Task Model

The focus of this paper is on periodic task sets with preemption thresholds. Each task τ_i is characterized by a 5-tuple of natural numbers denoted $(C_i, T_i, D_i, \pi_i, \gamma_i)$ with $C_i \leq D_i$, where C_i is the run time of task τ_i and D_i is its relative deadline; that is, after task τ_i is released for execution, it should complete within D_i time units. For periodic tasks, the period is denoted by T_i. Each task τ_i has a priority of π_i and a preemption threshold of γ_i. Consider the example shown in Table 1. This task set is the original example from Wang and Saksena's paper [13].

It was used to show that some task sets can be scheduled with preemption thresholds when no fixed priority assignment using purely preemptive or non-preemptive priority assignment will work. If each task has its preemption threshold set equal to its priority ($\gamma_i = \pi_i$ for all i), then purely preemptive scheduling results. On the other hand, if all preemption thresholds are set to the maximum priority, then non-preemptive scheduling is realized. In this way, preemption thresholds can be used to represent a wide range of scheduling strategies ranging from preemptive to non-preemptive.

Table 1. Periodic task set

i	C_i	T_i	D_i	π_i	γ_i
0	20	70	50	3	3
1	20	80	80	2	3
2	35	200	100	1	2

The goal of this paper is to develop a symbolic model to verify various properties and determine if the task set can be scheduled using different preemption thresholds. If the task set can be feasibly scheduled, then the model can also be used to obtain a set of minimal preemption thresholds; that is, preemption thresholds that are as small as possible. Next, we turn our attention to the input language used to develop the model using UPPAAL.

The core of the model input language is timed automata extended with data variables and tasks. Each edge of such extended automata can be labeled with three labels:

1. a guard containing a clock constraint and/or a predicate on data variables,
2. an action which can be an input or output action in the form of a! or a?, and
3. a sequence of assignments in the form: x = 0 when x is a real-valued clock or v = E when v is a data variable and E is a mathematical expression over data variables and constants.

A location of an extended automaton may be annotated with a task or a set of tasks that will be triggered when the transition leading to the location is taken. The triggered tasks will be put into a task queue, like the ready queue in an operating system, and scheduled to run according to a given scheduling policy. The scheduler should make sure that all task constraints are satisfied in scheduling the tasks in the task queue. To model concurrency and synchronization between automata, networks of automata are constructed in the standard way as in UPPAAL with the annotated sets of tasks on locations unioned [1, 7].

Four types of shared data variables can be used for communication and resource sharing:

1. Tasks can share variables with each other, and shared variables are protected by semaphores.
2. Tasks can read and update variables owned by the automata.
3. Automata can read (but not update) variables owned by the tasks.
4. Automata can share variables with each other.

In this paper, we limit the focus to periodic tasks that are independent and do not share resources. Next, we describe how schedulability analysis can be performed using UPPAAL [1].

A network of timed automaton annotated with tasks is considered as a design model. Given an extended automaton and a scheduling policy, the related schedulability analysis problem is to check whether there exists a reachable state of the scheduler automaton where a task misses its deadline. Such states are called non-schedulable states. An automaton is said to be non-schedulable with the given scheduling policy if it may reach a non-schedulable (ERROR) state. Otherwise, it is schedulable.

Consider the task set shown in Table 1. The highest priority task τ_0 (priority $\pi_0 = 3$) has a deadline of 50 and a period of 70. The lowest priority task τ_2 (priority $\pi_2 = 1$) has a deadline of 100 and a period of 200.

To check if a task set is schedulable, we construct a pair of timed automata – one to generate jobs to be released (called the PERIODIC_TASK automaton) and one to schedule the jobs released (called the SCHEDULER automaton). Then, a model checker is used to check the reachability of a predefined error state in the product automaton of the pair. If the error state is reachable, the task set is not schedulable.

In the next section, we describe a new two-clocks SCHEDULER automaton that can be used to test for the schedulability of task sets with preemption thresholds. It can also be used compute an optimal set of preemption thresholds if such an assignment exists.

3 Two-Clocks Model

The following two-clocks model has has fewer transitions than previous models. Also, unlike the encoding of the two-clocks scheduler by Fersman, et al. [7], there are five states in our model instead of four:

- $Idle_i$ - the ready job queue is empty,

- $Busy_i$ - jobs with priority greater than or equal to task i have arrived for execution,
- $Ready_i$ - the job in task i to be checked for feasibility has been released for execution, but has not started executing, and
- $Check_i$ - the job in task i to be checked for feasibility has started executing, and
- $Error_i$ - the checked job in task i missed its deadline.

The additional state, $Ready_i$, is needed because of the different way in which jobs may be preempted before or after being scheduled for execution when using preemption thresholds; after a task has been scheduled for execution, it can only be preempted by tasks with a priority greater than the running task's preemption threshold.

The first automaton, called PERIODIC_TASKS, is designed to model the releases of jobs within each task. The SCHEDULER automaton for the example given above is shown below in Figure 2. The following global declarations are used to specify the constraints shown in Table 1:

```
chan job[3];

const int C[3] = { 20, 20, 35 };
const int T[3] = { 70, 80, 200 };
const int D[3] = { 50, 80, 100 };
const int N[3] = { 40, 35, 14 };
const int PR[3] = { 3, 2, 1 };
const int PRT[3] = { 3, 3, 2 };
```

With periods of 70, 80, and 200, the hyperperiod is 2800. In one hyperperiod, there will be 40 jobs released in task τ_0, 35 in τ_1, and 14 in τ_2. The number of jobs over a hyperperiod are specified in N[3]. Finally, the priorities and preemption thresholds are specified in arrays PR[3] and PRT[3], respectively. Local declarations and parameters passed to the SCHEDULER automaton are shown below.

```
SCHEDULER(int P, int PT, int RT, int D, int id)
clock c, d;
int r;
int Cmax = 1;

int gt(int i, int x, int r)
{
  if (PR[i]>x)
    return (r+C[i]);
  else
    return (r);
}

int ge(int i, int x, int r)
{
  if (PRT[i]>=x)
    return (r+C[i]);
  else
    return (r);
}
```

Finally, the system is initialized using the following system declarations.

```
SYSTEM DECLARATIONS

S0 = SCHEDULER(PR[0],PRT[0],C[0],D[0],0);
S1 = SCHEDULER(PR[1],PRT[1],C[1],D[1],1);
S2 = SCHEDULER(PR[2],PRT[2],C[2],D[2],2);

system PERIODIC_TASKS,S0,S1,S2;
```

If we are only interested in testing for the worst case, then all schedulers can be initialized simultaneously. However, if we are testing other properties, it may be important to initialize the system with just one scheduler S0, S1, or S2.

The global channels, job[0], job[1], and job[2], are used by the PERIODIC_TASKS automaton to tell the SCHEDULER automaton that a new job from task τ_0, τ_1, or τ_2, respectively, has been released.

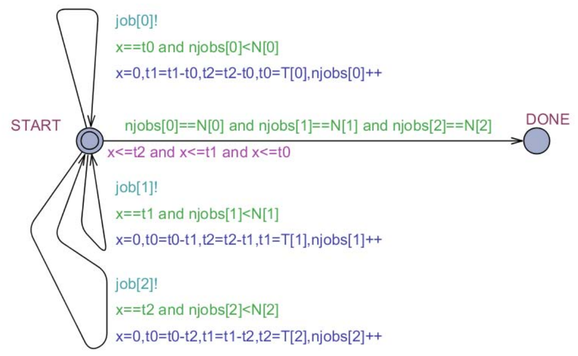

job[0]!
x==t0 and njobs[0]<N[0]
x=0,t1=t1-t0,t2=t2-t0,t0=T[0],njobs[0]++
START
njobs[0]==N[0] and njobs[1]==N[1] and njobs[2]==N[2]
DONE
x<=t2 and x<=t1 and x<=t0
job[1]!
x==t1 and njobs[1]<N[1]
x=0,t0=t0-t1,t2=t2-t1,t1=T[1],njobs[1]++
job[2]!
x==t2 and njobs[2]<N[2]
x=0,t0=t0-t2,t1=t1-t2,t2=T[2],njobs[2]++

Figure 1. PERIODIC_TASKS automaton

A local clock, x, is used to determine when the jobs are released. Local integers, t0, t1, and t2, are used to keep track of when the next job is to be released in each task, relative to the local clock x. Since, x is set back to 0 when each job is released, the release times are also updated then as well. Finally, the number of jobs released is recorded in njobs[i] for each task τ_i.

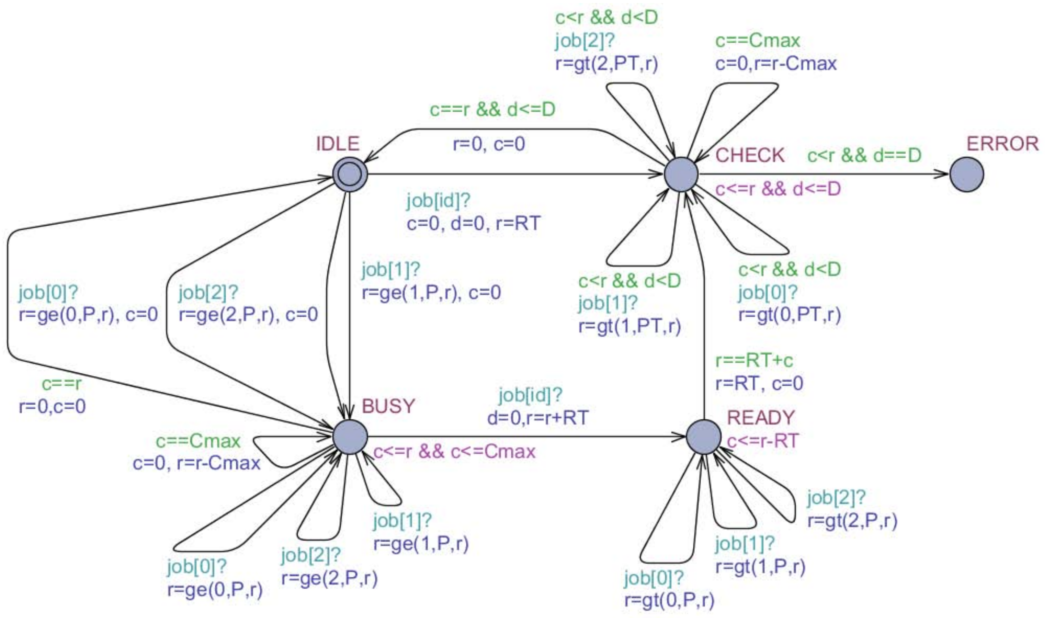


Figure 2. SCHEDULER automaton

```
PERIODIC_TASKS
clock x;
int[0,T[0]] t0;
int[0,T[1]] t1;
int[0,T[2]] t2;
int njobs[3];
```

The PERIODIC_TASKS automaton, shown in Figure 1, is essentially the same as the job generating automaton used to generate periodic jobs in previous models. The main differences are in the SCHEDULER automaton shown in Figure 2.

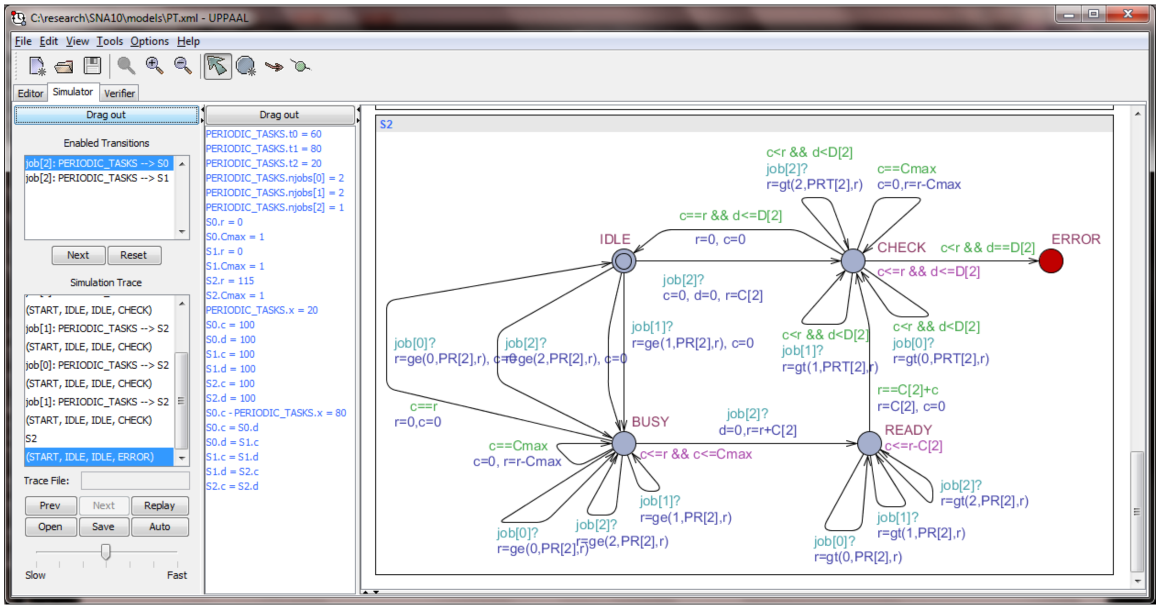
The intuition is the same as that used for traditional real-time response time analysis. In this model, the highest priority task is designated as task τ_0, and the lowest priority task is designated as task τ_2. When a message is received on channel job[i], a job has been released in task τ_i. Some job in task τ_i is nondeterministically selected to be checked for feasibility. The model checker will check all possible jobs in each task. The intuition behind the model is that if some job will miss its deadline, then all jobs in task τ_i that meet their deadlines will be processed in the BUSY state. When the job that will miss its deadline is released, a transition is taken to the READY state. When the job being analyzed is scheduled for execution, the scheduler enters the CHECK state. When the deadline is reached before the job has finished executing, a transition is taken to the ERROR state. If the first job will miss its deadline, then a transition can be taken directly from the IDLE state to the CHECK state. We rely on the model checker to test all possible jobs in each task, τ_i, to determine if there is a job that will miss its deadline.

Using the UPPAAL Verification Tool, a user can quickly check to see if the SCHEDULER ever enters the ERROR state by using the property: E<>(Si.ERROR) for i=0,1,2. If the propery is satisfied then, on some path, the SCHEDULER automaton eventually enters the ERROR state indicating that the task set is not schedulable. For the sample data given, using a deadline monotonic priority assignment, this property is satisfied. If some job misses its deadline, a trace can be generated and visualized in UPPAAL leading to the simulator output shown in Figure 3. Note that this is the output that results when purely preemptive scheduling is used; that is, $\gamma_i = \pi_i$ for all i. Note that γ_i is denoted as PRT[i] in the model.

```
const int PR[3] = { 3, 2, 1 };
const int PRT[3] = { 3, 2, 1 };
```

Recall that the priorities and preemption thresholds are specified in arrays PR[3] and PRT[3], respectively. To see that the SCHEDULER automaton S2 enters the ERROR state eventually on some path, we can verify that the property E<>(S2.ERROR) is satisfied, as shown below in Figure 4.

When the SCHEDULER automaton S2 goes from the CHECK state to the ERROR state, the value of S2.c = 100, but S2.r = 115 and S2.d = 100; that is, the first job in task τ_2 has run for 100 of 115 time units



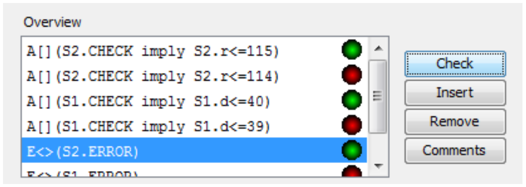
```
job[2]: PERIODIC_TASKS --> S0
job[2]: PERIODIC_TASKS --> S1
```

```
(START, IDLE, IDLE, CHECK)
job[1]: PERIODIC_TASKS --> S2
(START, IDLE, IDLE, CHECK)
job[0]: PERIODIC_TASKS --> S2
(START, IDLE, IDLE, CHECK)
job[1]: PERIODIC_TASKS --> S2
(START, IDLE, IDLE, CHECK)
S2
(START, IDLE, IDLE, ERROR)
```

```
PERIODIC_TASKS.t0 = 60
PERIODIC_TASKS.t1 = 80
PERIODIC_TASKS.t2 = 20
PERIODIC_TASKS.njobs[0] = 2
PERIODIC_TASKS.njobs[1] = 2
PERIODIC_TASKS.njobs[2] = 1
S0.r = 0
S0.Cmax = 1
S1.r = 0
S1.Cmax = 1
S2.r = 115
S2.Cmax = 1
PERIODIC_TASKS.x = 20
S0.c = 100
S0.d = 100
S1.c = 100
S1.d = 100
S2.c = 100
S2.d = 100
S0.c - PERIODIC_TASKS.x = 80
S0.c = S0.d
S0.d = S1.c
S1.c = S1.d
S1.d = S2.c
S2.c = S2.d
```

Figure 3. Missed deadline

needed to complete its execution when it misses its deadline at time 100 (D_2 = D[2] = 100). Even though it misses it's deadline, the worst-case response time is also shown to be 115 as shown below in Figure 4.

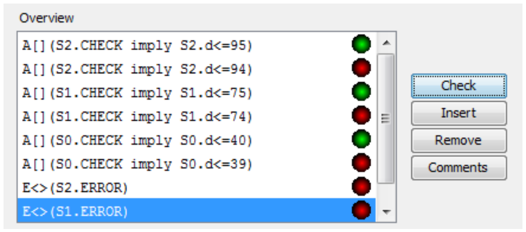


```
A[](S2.CHECK imply S2.r<=115)   ●
A[](S2.CHECK imply S2.r<=114)   ●
A[](S1.CHECK imply S1.d<=40)    ●
A[](S1.CHECK imply S1.d<=39)    ●
E<>(S2.ERROR)                   ●
E<>(S1.ERROR)                   ●
```

Figure 4. Preemptive tasks output

If we increase the preemption thresholds of the tasks as shown in Table 1, then the task set is schedulable. In this case, task τ_2 the low priority task, has a deadline of 100 and a period of 200. This can be easily done in the model by simply changing the constants in the global declarations; that is, changing the PRT[3] array declaration to const int PRT[3] = { 3, 3, 2 }.

Using the UPPAAL Verification Tool, it is possible to compute the worst-case response time for each task that meets its deadline using the property: A[](S2.CHECK imply (S2.d<=95)) which is satisfied. The same property with an upper bound of 94 is not satisfied, as shown in Figure 5. Thus, the worst-case response time of task τ_2 is 95, which is less than it's deadline of 100.



```
A[](S2.CHECK imply S2.d<=95)   ●
A[](S2.CHECK imply S2.d<=94)   ●
A[](S1.CHECK imply S1.d<=75)   ●
A[](S1.CHECK imply S1.d<=74)   ●
A[](S0.CHECK imply S0.d<=40)   ●
A[](S0.CHECK imply S0.d<=39)   ●
E<>(S2.ERROR)                  ●
E<>(S1.ERROR)                  ●
```

Figure 5. Verification output

In addition to scheduling properties, it is possible analyze other safety and liveness properties using the model checker UPPAAL.

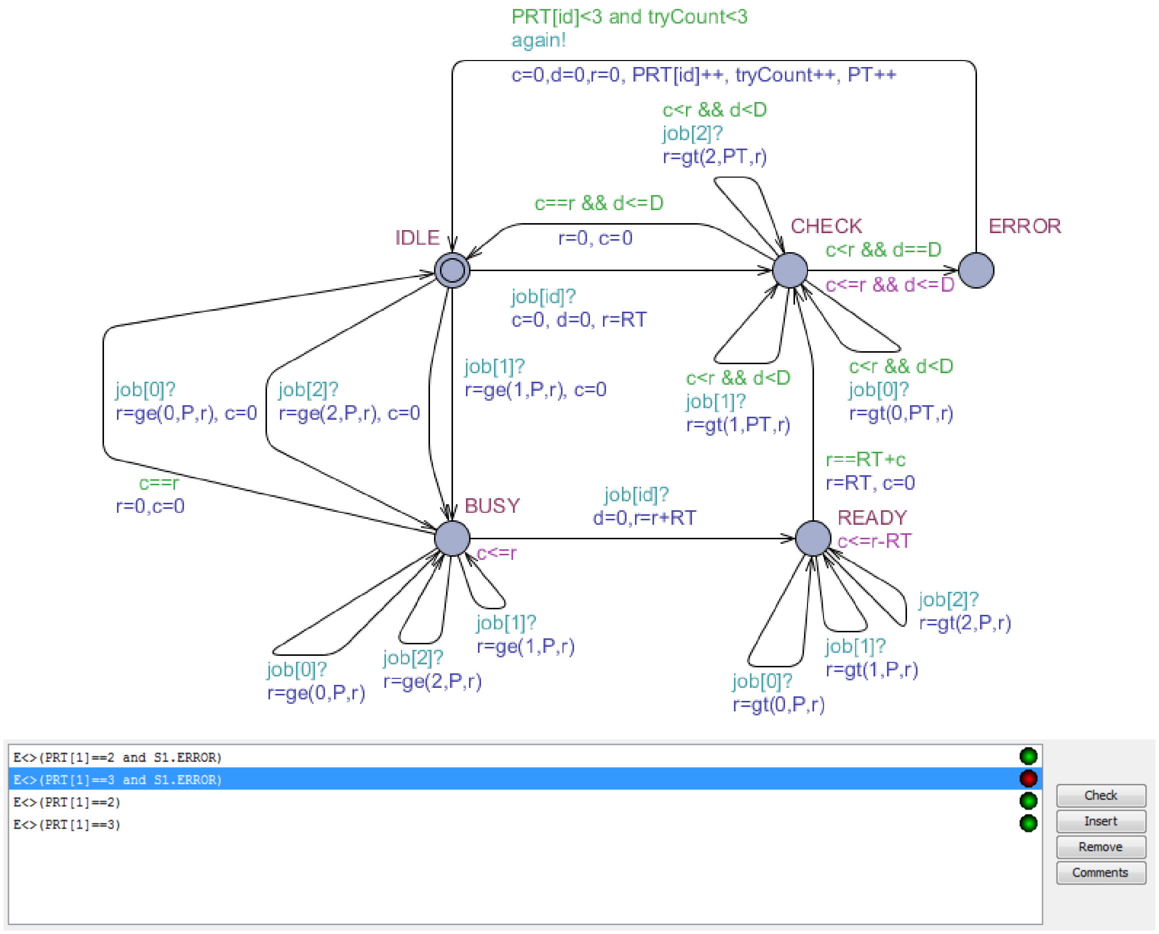
As noted in [6], the value used for Cmax to decrement r can be any positive value. It turns out that if Cmax is larger than the level-i busy period, then the number of states generated and searched will be the least possible, but the value of r may reach the length of the level-i busy period.

The number of states searched for each value of Cmax is shown below in Table 2. This is based on the case when the preemption thresholds are set to 3, 3, and 2, respectively.

Table 2. Bounds on number of states.

Cmax	States	Cmax	States
1	150,180	50	1,793
3	20,820	100	1,635
6	8,155	250	1,617
12	3,805	500	1,617

If we remove the transitions involving Cmax, then the number of states searched and stored is 1,617. The



```
E<>(PRT[1]==2 and S1.ERROR)
E<>(PRT[1]==3 and S1.ERROR)
E<>(PRT[1]==2)
E<>(PRT[1]==3)
```

Figure 6. Updated SCHEDULER automaton

number of states is directly related to the amount of time required to evaluate the property.

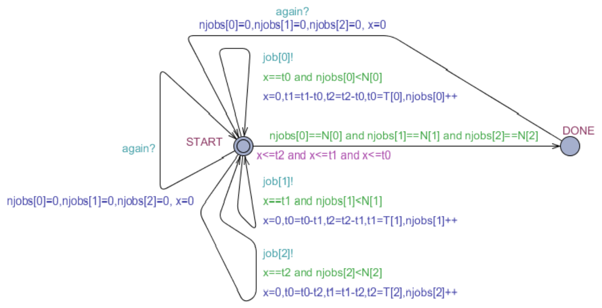

again?
njobs[0]=0,njobs[1]=0,njobs[2]=0, x=0
job[0]!
x==t0 and njobs[0]<N[0]
x=0,t1=t1-t0,t2=t2-t0,t0=T[0],njobs[0]++
again?
START
njobs[0]==N[0] and njobs[1]==N[1] and njobs[2]==N[2]
x<=t2 and x<=t1 and x<=t0
DONE
njobs[0]=0,njobs[1]=0,njobs[2]=0, x=0
job[1]!
x==t1 and njobs[1]<N[1]
x=0,t0=t0-t1,t2=t2-t1,t1=T[1],njobs[1]++
job[2]!
x==t2 and njobs[2]<N[2]
x=0,t0=t0-t2,t1=t1-t2,t2=T[2],njobs[2]++

Figure 7. Updated PERIODIC_TASKS automaton

Finally, if we set the preemption thresholds to be as large as possible, then non-preemptive scheduling is realized; e.g., set const int PRT[3] = 3,3,3. UPPAAL queries can be used to verify that the highest priority task misses its deadline.

The scheduler and job creation automata can be modified to find an optimal assignment of preemption thresholds that are minimal (preemption thresholds as small as possible). To find an optimal assignment that is minimal, the SCHEDULER automaton shown in Figure 6 can be used. If the current assignment leads to an error, then the preemption threshold is incremented, and the automaton is reset by sending a message on the *again* channel to try again as long as the preemption threshold is less than the maximum priority. The SCHEDULER automaton goes from the ERROR to the IDLE state, and the PERIODIC_TASKS automaton goes back to the START state, and starts sending the same set of jobs all over again.

For the second task in our example, the threshold must be elevated to the maximum priority 3 to ensure that the task set is feasible. As shown in Figure 6, if the preemption threshold of task 1 is initialized to 2, then the ERROR state is reachable, but if the preemption threshold is initialized as 3, then the ERROR state is not reachable. Similarly, the SCHEDULER automaton could be adapted to compute the maximum feasible preemption threshold, in most cases we are only interested in finding the smallest possible preemption threshold that will work.

4 Conclusions

This paper presented a simple model that can be used to analyze the feasibility of periodic task sets with preemption thresholds and compute an optimal assignment of preemption thresholds. The model consists of a set of two automata: a PERIODIC_TASKS automaton to model the release of periodic jobs, and a SCHEDULER automaton to model the scheduler. The model generated can be used with UPPAAL 4.0 to determine if the task set is feasible, and to compute the worst-case response times of tasks that meet their deadlines. If the task set is not feasible, the model can also be used to identify why the task set fails to meet its deadlines. The example UPPAAL models and queries generated are available on-line at: `http://www.cis.ksu.edu/~neilsen/PDPTA11/`.

The model and queries could easily be generalized to test the feasibility of other periodic task sets with arbitrary start times and arbitrary deadlines. UPPAAL is a very powerful tool that can be used for many types of verification for distributed and real-time systems. An interesting next step will be to incorporate dynamic voltage scheduling into the model and verify some of the properties specified in [8].

References

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *In Proc. of FORMATS03, number 2791 in LNCS*, pages 60–72. Springer-Verlag, 2003.

[2] G. Behrmann, A. David, K.G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems (QEST) 2006*, IEEE Computer Society, pages 125–126, 2006.

[3] J. Bengtsson, F. Larsson, P. Pettersson, W. Yi, P. Christensen, J. Jensen, P. Jensen, K. Larsen, and T. Sorensen. Uppaal: a tool suite for validation and verification of real-time systems, 1996.

[4] J. Chen, A. Harji, and P. Buhr. Solution space for fixed-priority with preemption threshold. In *11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS 05)*, pages 385–394, 2005.

[5] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: schedulability and decidability. In *In Proceedings of TACAS 2002*, pages 67–82. Springer-Verlag, 2002.

[6] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis using two clocks. In *In 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, pages 224–239. Springer, 2003.

[7] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354(2):301 – 317, 2006. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003).

[8] R. Jejurikar and R. Gupta. Integrating preemption threshold scheduling and dynamic voltage scaling for energy efficient real-time systems. In *International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 04)*, 2004.

[9] M.L. Neilsen. A generalized token-based mutual exclusion algorithm for wireless networks. In *Proceedings of the 20th Int'l Conference on Parallel and Distributed Computing Systems (PDCS-2007), Las Vegas, Nevada, USA*, ISCA, 2007.

[10] M.L. Neilsen. Model checking token-based distributed mutual exclusion algorithms. In *Proceedings of the 15th International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 10–16, 2009.

[11] M.L. Neilsen. Symbolic schedulability analysis of task sets with arbitrary deadlines. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, 2010.

[12] M.L. Neilsen. Symbolic schedulability analysis of task sets with preemption thresholds. In *Second International Sensor Networks and Applications Conference (SNA'10)*, 2010.

[13] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *RTCSA*, pages 328–, 1999.