Task Automata: Schedulability, Decidability and Undecidability

Elena Fersman [1], Pavel Krcal, Paul Pettersson [2] and Wang Yi [3]

Email: {pavelk,paupet,yi}@it.uu.se
Department of Information Technology
Uppsala University

Abstract

We present a model, *task automata*, for real time systems with non-uniformly recurring computation tasks. It is an extended version of timed automata with asynchronous processes that are computation tasks generated (or triggered) by timed events. Compared with classical task models for real time systems, task automata may be used to describe tasks (1) that are generated non-deterministically according to timing constraints in timed automata, (2) that may have interval execution times representing the best case and the worst case execution times, and (3) whose completion times may influence the releases of task instances. We generalize the classical notion of schedulability to task automata. A task automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events generated by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines. Our first technical result is that the schedulability for a given scheduling strategy can be checked algorithmically for the class of task automata when the best case and the worst case execution times of tasks are equal. The proof is based on a decidable class of suspension automata: timed automata with bounded subtraction in which clocks may be updated by subtractions within a bounded zone. We shall also study the borderline between decidable and undecidable cases. Our second technical result shows that the schedulability checking problem will be undecidable if the following three conditions hold: (1) the execution times of tasks are intervals, (2) the precise finishing time of a task instance may influence new task releases, and (3) a task is allowed to preempt another running task.

Key words: Real Time Systems, Schedulability Analysis, Timed Automata, Modeling and Verification

1 Introduction

One of the most important issues in developing real time systems is *schedulability analysis* prior to implementation. In the area of real time scheduling, there are well-studied methods [1], e.g., rate monotonic scheduling, that are widely applied in the analysis of periodic tasks with deterministic behaviours. For *non-periodic* tasks with non-deterministic behaviours, there are no satisfactory solutions. There are approximative methods with pessimistic analysis, e.g., using periodic tasks to model sporadic tasks when control structures of tasks are not considered. The advantage of automata-theoretic approaches, e.g., using timed automata in modeling systems is that one may specify general timing constraints on events and model other behavioural aspects such as concurrency and synchronization. However, it is not clear how timed automata can be used for schedulability analysis because there is no support for specifying resource requirements and hard time constraints on computations, e.g., deadlines.

Following the work of [2], we study task automata, that are timed automata extended with real time tasks triggered by events. A task is an executable program characterized by its best case and worst case execution time, deadline, and possibly other parameters such as priorities for scheduling. The main idea is to associate each location of a timed automaton with a task (or a set of tasks in the general case). Intuitively a discrete transition leading to a location in the automaton denotes an event triggering an instance of the annotated task and the guard (clock constraints) on the transition specifies the possible arrival times of the event. Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of a running task (with the highest priority) and idling for the other tasks. Discrete transitions correspond to the arrival of new task instances. Whenever a task is triggered, it will be put into the scheduling queue for execution (i.e., the ready queue in operating systems). We assume that the tasks will be executed according to a given scheduling strategy, e.g., FPS (fixed priority scheduling) or EDF (earliest deadline first).

For example, consider the automaton shown in Figure 1. It has three locations l_0, l_1, l_2, and two tasks P and Q (triggered by a and b) with interval computation times $[1, 2]$ and $[2, 4]$ (the best case and the worst case execution times), and relative deadlines 10 and 8, respectively. The automaton models

[1] Current address: Ericsson AB, Torshamnsgatan 23, SE-164 80 Stockholm, Sweden. Email: elena.fersman@ericsson.com.
[2] Current address: Department of Computer Science and Electronics, Mälardalen University, Sweden. Email: Paul.Pettersson@mdh.se.
[3] Corresponding author: Wang Yi, Box 337, 751 05, Uppsala, Sweden. Email: yi@it.uu.se. Tel: +46 18 4713110.

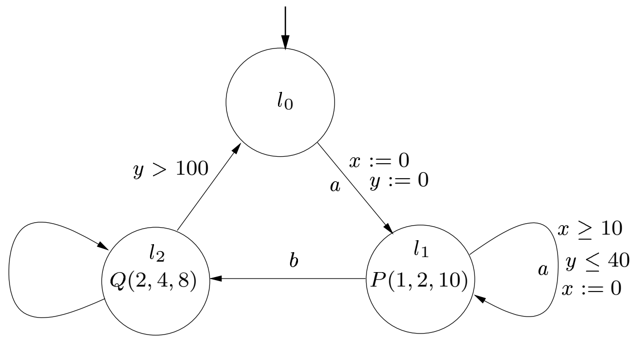

Fig. 1. A task automaton.

a system starting in l_0 that may move to l_1 by event a at any time. This triggers the task P. In l_1, as long as the constraints $x \geq 10$ and $y \leq 40$ hold, when an event a occurs an instance of task P will be created and put into the scheduling queue. However, it cannot create more than 5 instances of P in l_1, because the constraint $y \leq 40$ will be violated after 40 time units. In fact, every instance will be computed before the next instance arrives and the scheduling queue may contain at most one task instance. Therefore, no task instance of P will miss its deadline. The system is also able to accept b, switch from l_1 to l_2, and trigger Q. Because there are no constraints labeled on the b-transition in l_2, it may accept any number of b's and create any number of Q's in zero time. However, after more than two copies of Q, the queue will be non-schedulable i.e. a deadline may be violated. This means that the system is non-schedulable. Thus, zeno behaviours will correspond to non-schedulability, which is a natural property of the model.

We shall formalize the notion of schedulability in terms of reachable states. A state of a task automaton will be a triple (l, u, q) consisting of a location l, a clock valuation u, and a task queue q. The task queue contains pairs of remaining computation times and relative deadlines for all released tasks. A scheduling strategy is a function on queues, which inserts a new task instance into the task queue according to the task parameters such as fixed priorities, remaining computation times, and (or) deadlines. Formally, we assume that a scheduling strategy can be encoded as timed automata. We shall see that the existing scheduling strategies (preemptive or non-preemptive) in the literature such as EDF or FPS satisfy this condition. An automaton is schedulable if there exists a scheduling strategy with which all tasks in q can be computed within their deadlines for all reachable states (l, u, q) of the automaton.

In [2], it is shown that under the assumption that the tasks are non-preemptive, the schedulability checking problem for a given (non-preemptive) scheduling strategy can be transformed to a reachability problem for ordinary timed automata and thus it is decidable. For *preemptive scheduling* strategies, it has been suspected that the schedulability checking problem is undecidable because in preemptive scheduling we must use stopwatches to accumulate computation times for tasks. In this paper, we show that to model the scheduling problems, the expressive power of stopwatch automata is not needed.

Our main technical result is that the schedulability checking problem related to a preemptive scheduling strategy is decidable for a large class of task automata. We show that the problem is decidable if the best case and the worst case computation times of tasks are equal. The crucial observation in the proof is that the schedulability checking problem can be translated to a reachability problem for a decidable class of suspension automata [3] – timed automata with bounded subtraction where clocks may be updated with subtraction only in a bounded zone. We also show that the case with variable execution times can be reduced to the previous one for FPS and EDF if the finishing times of tasks do not influence release times of new tasks.

In particular, the schedulability problem related to EDF can be checked for these classes of automata. EDF is optimal in the sense that if it cannot schedule a task queue, no other scheduling strategy can. Therefore, the general schedulability checking problem for a task automaton (whether there is a strategy which can schedule the automaton) can be checked if the best case and the worst case computation times of tasks are equal or if the finishing times of tasks do not influence release times of new tasks.

We shall also study the borderline between decidable and undecidable cases. It is shown that the schedulability problem for preemptive scheduling strategies is undecidable if task execution times may vary within an interval representing the best and the worst case execution times. This is a surprisingly negative result that such a subtle difference can turn a decidable problem to an undecidable one. More precisely, the schedulability problem for many scheduling strategies is undecidable if these three conditions hold: (1) the execution times of tasks are intervals, (2) the precise finishing time of a task may influence the new task releases and (3) a task is allowed to preempt another running task.

The rest of this paper is organized as follows: Section 2 describes the notions of task and scheduling strategy, and the syntax and semantics of task automata. Section 3 describes scheduling problems related to task automata with a summary on decidability and undecidability results. Section 4 is devoted to proofs for the decidable cases. Section 5 presents a proof for the undecidable case. Section 6 concludes the paper with summarized results and future work, and a brief summary on and comparison with related work.

2 Timed Automata with Tasks

We extend timed automata with asynchronous processes, i.e., tasks triggered by events and computed asynchronously. The main idea is to associate each location of a timed automaton with an abstraction of an executable program called a *task type* or simply a task.

2.1 Tasks and Scheduling Strategy

Assume a set of *task types* $\mathcal{P}$ ranged over by P, Q, R, etc. A task type or simply a task may have task parameters such as fixed priority, computation time (the best and the worst case), deadline, and resource requirements, e.g., on memory consumption. For simplicity, we do not consider resource requirements in this paper. Assume that B, W, D are three natural numbers such that $B \leq W \leq D$ and $0 < W$. A task type is a tuple (P, B, W, D), written $P(B, W, D)$, where P is the task name, B is the *best case execution time*, W is the *worst case execution time* and D is the *relative deadline*. Note that D is a relative deadline meaning that whenever an instance of P is released, it should be computed within D time units. We sometimes say that the computation time of P is in $[B, W]$. When a set of tasks is scheduled according to fixed priorities then we assume that each task type has assigned a priority.

A task may have several instances that are different copies of the same program. A task instance is a tuple (P, b, w, d), written $P(b, w, d)$, where P is a task name, $b \in \mathbb{R}$ is a best case remaining computation time, $w \in \mathbb{R}$ is a worst case remaining computation time, and $d \in \mathbb{R}$ is a relative deadline. We shall use p_i to denote a task instance and, without confusion, p_i's task type will be understood as (P_i, B_i, W_i, D_i). Note that different task instances may be of the same task type with the same task parameters. A task queue is a list of task instances denoted $[P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$. The discrete part of a queue is a list containing only the task names. A set of all task queues containing instances of the task types from $\mathcal{P}$ is denoted $Q_{\mathcal{P}}$.

We shall study scheduling problems for single-processor systems. Thus we assume that a task queue is a sorted list whose head element is the task instance running on the processor, and the other ones are waiting. A scheduling strategy, e.g., FPS (fixed priority scheduling), SJF (shortest job first), or EDF (earliest deadline first), is a function which inserts released tasks into the task queue.

More precisely, a *scheduling strategy* (or scheduling function) $\mathsf{Sch} : \mathcal{P} \times Q_{\mathcal{P}} \mapsto Q_{\mathcal{P}}$ is a function which given a task instance and a task queue returns a task queue with the task instance inserted and the order of the other task instances preserved. For example, $\mathsf{EDF}(P(1,3,10),[Q(3,4,5.3),R(0,2,19)]) = [Q(3,4,5.3),P(1,3,10),R(0,2,19)]$. A scheduling strategy has to satisfy the following condition. The decision on where the new task instance is inserted in the queue can be made only by comparing the task parameters of the new task instance with each of the existing instances in the queue and by considering the discrete part of the queue. The task parameters are either the remaining best and worst case computation times, or the remaining relative deadlines. We formalize the concept of a scheduling strategy in terms of timed automata in Definition 7.

Scheduling strategies may be preemptive or non-preemptive:

(1) A non-preemptive strategy will never insert the new task as the first element of the queue.
(2) A preemptive strategy may insert the new task in the first position if its task type is different from the current running task and all suspended (preempted) tasks in the queue.

To talk about computation and resource consumption, we shall use a function $\mathsf{Run} : Q_{\mathcal{P}} \times \mathbb{R}_{\geq 0} \mapsto Q_{\mathcal{P}}$ which given a real number t and a task queue q returns the task queue after t time units of execution on a processor. The result of $\mathsf{Run}(q,t)$ for $t \leq w_1$ and $q = [P(b_1,w_1,d_1), Q(b_2,w_2,d_2), \ldots, R(b_n,w_n,d_n)]$ is defined as $q' = [P(b_1-t, w_1-t, d_1-t), Q(b_2,w_2,d_2-t), \ldots, R(b_n,w_n,d_n-t)]$. For example, let $q = [Q(2,3,5), P(4,7,10)]$. Then $\mathsf{Run}(q,3) = [Q(-1,0,2), P(4,7,7)]$ in which the first task has been executed for 3 time units (and it will be removed from the queue).

A task instance $P(b,w,d)$ in the queue may finish when $b \leq 0$ and $w \geq 0$, and it must finish when $w = 0$. Finished tasks are removed from the queue. This is reflected in the definition of semantics of task automata, Definition 2.

2.2 Task Automata

As in timed automata, assume a finite alphabet $\mathcal{A}ct$ ranged over by $a, b, \ldots$ and a finite set of real-valued clocks $\mathcal{C}$ ranged over by $x_1, x_2, \ldots$. We use $\mathcal{B}(\mathcal{C})$ ranged over by g to denote the set of conjunctive formulas of atomic constraints in the form: $x_i \bowtie C$ or $x_i - x_j \bowtie C$ where $x_i, x_j \in \mathcal{C}$ are clocks, $\bowtie \in \{\leq, <, \geq, >\}$, and C is a natural number. The elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints*.

Assume a distinguished clock x_{done} which is reset every time a task finishes its computation and is removed from the task queue. This clock can be used to model data dependencies or precedence relations between tasks. One may introduce such a clock or a boolean for every task type without changing the technical results of this paper. It can be easily seen from the decidability proofs that more such clocks or boolean variables can be accommodated and that it is enough to have one clock for the undecidability result. Therefore, for simplicity of presentation, we use only x_{done}.

Definition 1 *A task automaton over actions $\mathcal{A}ct$, clocks $\mathcal{C}$, and task types $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M, x_{done}\rangle$ where*

- *N is a finite set of locations ranged over by l, m, n,*
- *$l_0 \in N$ is the initial location,*
- *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \mathcal{A}ct \times 2^{\mathcal{C}} \times N$ is the set of edges,*
- *$I : N \mapsto \mathcal{B}(\mathcal{C})$ is a function assigning each location with a clock constraint (a location invariant),*
- *$M : N \hookrightarrow \mathcal{P}$ is a partial function assigning locations with task types,*[4] *and*
- *$x_{done} \in \mathcal{C}$ is the clock which is reset whenever a task finishes.*

When $\langle l, g, a, r, l'\rangle \in E$, we write $l \xrightarrow{g\, a\, r} l'$.

A task automaton is said to have no task feedback *if none of its guards or invariants contains the clock x_{done}. Further, a task automaton has* fixed computation times *of tasks if $B = W$ for all task types $P(B, W, D)$.*

Note that the distinguished clock x_{done} may be used in the guards or invariants of a task automaton, which means that the finishing times of tasks may influence the behaviour of the task automaton. However, the finishing time of a task does not have any influence on the behaviour of a task automaton when it has no task feedback.

2.3 Operational Semantics

Similarly to timed automata, a task automaton may perform two types of transitions. Delay transitions correspond to the execution of the running task (e.g., a task with the highest priority or with the earliest deadline) and idling for the other tasks waiting to run. These transitions are split into three subtypes according to the queue status (empty, non-empty) and the fact, whether a task finishes. Discrete transitions correspond to the arrivals of new task instances.

We represent the values of clocks as functions (i.e., clock assignments) from $\mathcal{C}$ to the non–negative reals $\mathbb{R}_{\geq 0}$. We denote by $\mathcal{V}$ the set of clock assignments for $\mathcal{C}$. Naturally, a semantic state of an automaton is a triple (l, u, q) where l is the current location, $u \in \mathcal{V}$ denotes the current values of clocks, and q is the

[4] Note that M is a partial function meaning that some of the locations may have no tasks.

current task queue. By u_0 we denote a clock assignment such that $u_0(x) = 0$ for all clocks x.

We use $u \models g$ to denote that the clock assignment u satisfies the constraint g. For $t \in \mathbb{R}_{\geq 0}$, we use $u + t$ to denote the clock assignment which maps each clock x to the value $u(x) + t$, and $u[r]$ for $r \subseteq \mathcal{C}$, to denote the clock assignment which maps each clock in r to 0 and agrees with u for the other clocks (i.e., $\mathcal{C} \backslash r$). Now we are ready to present the operational semantics for task automata as labeled transition systems (LTS).

Definition 2 *Given a scheduling strategy* Sch*, the semantics of an automaton* $A = \langle N, l_0, E, I, M, x_{done}\rangle$ *is a labeled transition system* $[\![A_{\mathsf{Sch}}]\!]$ *with an initial state* $(l_0, u_0, [])$ *and transitions defined by the following rules:*

- $(l, u, q) \xrightarrow{a}_{\mathsf{Sch}} (l', u[r], \mathsf{Sch}(M(l'), q))$ *if* $l \xrightarrow{g\, a\, r} l'$, $u \models g$, *and* $u[r] \models I(l')$,
- $(l, u, []) \xrightarrow{t}_{\mathsf{Sch}} (l, u + t, [])$ *if* $t \in \mathbb{R}_{\geq 0}$ *and* $(u + t) \models I(l)$,
- $(l, u, P(b, w, d) :: q) \xrightarrow{t}_{\mathsf{Sch}} (l, u + t, \mathsf{Run}(P(b, w, d) :: q, t))$ *if* $t \in \mathbb{R}_{\geq 0}, t \leq w$ *and* $(u + t) \models I(l)$, *and*
- $(l, u, P(b, w, d) :: q) \xrightarrow{fin}_{\mathsf{Sch}} (l, u[x_{done}], q)$ *if* $b \leq 0 \leq w$ *and* $u[x_{done}] \models I(l)$,

where $P(b, w, d) :: q$ *denotes the queue with the task instance* $P(b, w, d)$ *inserted into* q *(at the first position),* $[]$ *denotes the empty queue, and* $fin \notin \mathcal{A}ct$ *is a distinct action name.*

Note that the transition rules are parameterized by Sch (scheduling strategy). Whenever it is understood from the context, we shall omit Sch from the transition relation. We have the same notion of reachability as for timed automata.

Definition 3 *We shall write* $(l, u, q) \longrightarrow_{\mathsf{Sch}} (l', u', q')$ *if either* $(l, u, q) \xrightarrow{a}_{\mathsf{Sch}} (l', u', q')$ *for an action* a*, or* $(l, u, q) \xrightarrow{t}_{\mathsf{Sch}} (l', u', q')$ *for a delay* t*, or* $(l, u, q) \xrightarrow{fin}_{\mathsf{Sch}} (l', u', q')$*. For a task automaton with initial state* $(l_0, u_0, [])$ *and a scheduling strategy* Sch*,* (l, u, q) *is reachable iff* $(l_0, u_0, []) \longrightarrow^*_{\mathsf{Sch}} (l, u, q)$*.*

Clearly task automata are at least as expressive as timed automata. In fact, $\langle N, l_0, E, I\rangle$ is an ordinary timed automaton. Intuitively, a discrete transition of the automaton denotes an event triggering a task annotated in the target location, and the guard on the edge specifies all the possible arrival times of the event (or the annotated task). Whenever a task is triggered, it will be put into the scheduling (or task) queue for execution (corresponding to the ready queue in operating systems). In general, the task queue is unbounded though the constraints of a given automaton may restrict the possibility of reaching states with infinitely many different task queues. For example, we may model time-triggered periodic tasks as a simple automaton as shown in Figure 2(a) where P is a periodic task with computation time in $[1, 2]$, deadline 8 and period 20. More generally, it may model systems containing both periodic and sporadic tasks as shown in Figure 2(b) which is a system consisting of 4 tasks as annotation on locations, where P and Q are triggered by time every 20 and 40 time units respectively (specified by the constraints: $x = 20$ and $x = 40$), and R and S are sporadic or event driven by event a and b respectively.

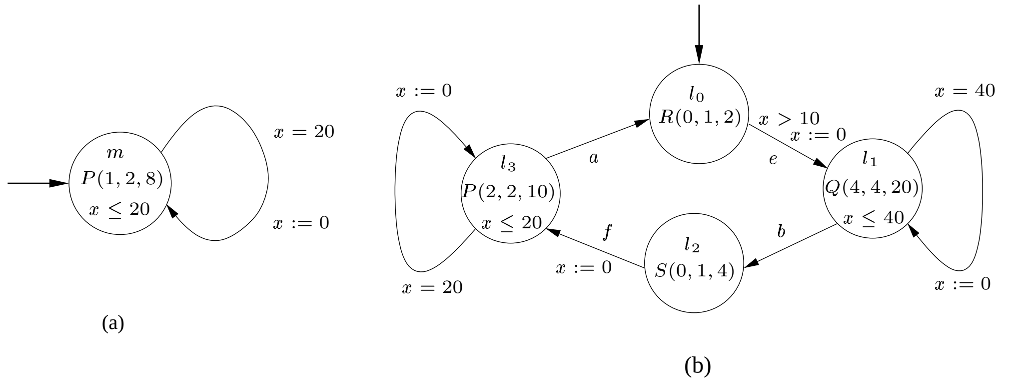

Fig. 2. Modeling Periodic and Sporadic Tasks.

To handle concurrency and synchronization, a parallel composition of task automata may be defined as a product automaton in the same way as for ordinary timed automata (e.g., see [4]). The only difference is that a product location might be assigned more than one task. In such a case, auxiliary locations are introduced so that each of them has at most one task assigned, the tasks are released at the same timepoint, and all interleavings (all orders in which the tasks are released) are allowed. Note that the parallel composition here is only an operator to construct models of systems based on their components. It has nothing to do with multi-processor scheduling.

As standard timed automata, the underlying timed automata of task automata may have time converging behaviours known as *timestops*, that are sequences of transitions where time does not diverge. Consider, for example, a location with no outgoing transitions and an invariant $x < 10$ for a clock x. Clearly, the automaton cannot delay in this location for longer than 10 time units. This would imply that some tasks in the queue never finish their computation if it requires more than 10 time units. Also, as time is not progressing, no deadline will be violated either. In order to simplify technical details in the proofs we prohibit timestops caused by timed automata invariants. However, this is not fundamental for our results and it is not difficult to check that it is also possible to handle a situation when the time does not progress. We refer to the standard semantics ([5]) of timed automata in the next definition.

Definition 4 *A task automaton $A = \langle N, l_0, E, I, M, x_{done}\rangle$* has no timestops *if for the underlying timed automaton $A' = \langle N, l_0, E, I\rangle$, for any state (l, u) of A', and all $L \in \mathbb{R}_{\geq 0}$ there exists a path $(l, u) \longrightarrow^* (l', u')$ in the semantics LTS of A' such that the sum of the time delays on the path from (l, u) to (l', u') is equal to L.*

From now on, we assume that task automata have no timestops which implies that each task will be eventually computed in such automata.

To demonstrate the semantics of task automata, consider the automaton in Figure 2(b). Assume that preemptive earliest deadline first (EDF) strategy is used to schedule the task queue. For example, the automaton may stay in l_1 or l_2 where instances of the periodic tasks P and Q may be released and computed. Or it may loop from l_0 back to l_0 through l_1, l_2, and l_3. Though the queue is growing during these transitions, the generation of new task instances will be slowed down by the constraint $x > 10$ labeled on the edge from l_0 to l_1 and the queue will be reduced by delay transitions, e.g.,

$$(l_0, [x = 0, x_{done} = 0], [S(-0.3, 0.7, 3.7), R(0, 1, 2), P(2, 2, 10), Q(3.5, 3.5, 19.2)]) \longrightarrow^+ (l_0, [x = 10, x_{done} = 3.1], []).$$

A question of interest is whether it can perform a sequence of transitions leading to a state where a deadline has been missed.

3 Schedulability Analysis

In this section we study verification problems related to the model presented in the previous section. One of the most interesting properties of task automata related to the task queue is schedulability.

3.1 Schedulability of Task Automata

As all deadlines in task automata are hard, we define schedulability for a given scheduling strategy as impossibility of reaching a state where some deadline is missed. We use q_{err} to denote queues containing a task instance $P(b, w, d)$ with $d < 0$.

Definition 5 *(Schedulability) A task automaton A with initial state $(l_0, u_0, [])$ is non-schedulable with* Sch *if $(l_0, u_0, []) \longrightarrow^*_{\mathsf{Sch}} (l, u, q_{err})$ for some l and u. Otherwise, we say that A is schedulable with* Sch*. More generally, we say that A is schedulable if and only if there exists a scheduling strategy* Sch *with which A is schedulable.*

We also need a notion of non-schedulable queues that are queues which will inevitably lead to a deadline miss with time progress (if all tasks take their worst case computation times).

Definition 6 *A queue* $q = [P_1(b_1, w_1, d_1) \ldots P_n(b_n, w_n, d_n)]$ *is* non-schedulable *if* $w_1 + \cdots + w_i > d_i$ *for some* $1 \leq i \leq n$.

Even though queues are not bounded in general, an important observation is that all schedulable queues are bounded. First, note that a task instance that has been started cannot be preempted by another instance of the same task type. This means that there is only one instance of each task type in the queue whose computation time can be a real number and it can be arbitrarily small. Thus the number of instances of each task type $P_i \in \mathcal{P}$, in a schedulable queue is bounded by $\lceil D_i/W_i \rceil$ and the size of schedulable queues is bounded by $\sum_{P_i \in \mathcal{P}} \lceil D_i/W_i \rceil$. This is an important property of our model, because it allows us to code schedulability checking problems as reachability problems.

Throughout of the paper, we shall distinguish three situations according to the queue status:

(1) A queue is an *error-queue* denoted q_{err} if a deadline is already missed.
(2) A queue is *non-schedulable* as defined in Definition 6 if it will inevitably evolve to an error-queue.
(3) A queue is *overflowed* if it contains more than $\lceil D_i/W_i \rceil$ instances of P_i for some i.

Note that an overflowed queue is definitely non-schedulable. But a very short (not overflowed) queue can also be non-schedulable. We shall say that a state is *adequate* if its queue is neither an overflowed queue nor an error-queue. However, an adequate state may contain a non-schedulable queue.

Before presenting the results, we formalize the concept of scheduling strategy used in this paper in terms of timed automata. Assume a task type $P(B, W, D)$ and a task queue $[P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$. We construct a diagonal-free (i.e. clocks are compared only to constants) timed automaton with clocks $x_{b_1}, x_{w_1}, x_{d_1}, \ldots, x_{b_n}, x_{w_n}, x_{d_n}$, $n+2$, locations $l_0, l_1, l_2, \ldots l_{n+1}$, and $n+1$ edges from l_0 to l_i for $i \leq n+1$. We call such an automaton a *decision automaton.*

Definition 7 *A* scheduling strategy $\mathsf{Sch} : \mathcal{P} \times Q_{\mathcal{P}} \mapsto Q_{\mathcal{P}}$ *is a function satisfying the following condition. For each task type* $P(B, W, D)$ *and a task queue* $[P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$, *one can effectively construct a decision automaton such that* $\mathsf{Sch}(P(B, W, D), [P_1\ (b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)])$ *inserts* $P(B, W, D)$ *into the queue at the* k*-th position if and only if* l_k *is the only location reachable from* (l_0, u) *where* $u(x_{b_i}) = b_i, u(x_{w_i}) = w_i, u(x_{d_i}) = d_i$ *for all* $1 \leq i \leq n$.

Note that to make location l_k reachable, the automaton should be constructed in such a way that the edges from l_0 to l_i for all $i \leq n+1$ are labeled with guards corresponding to the conditions on the task types and parameters to be checked for the scheduler in making the decision on where the new task instance, i.e., $P(B, W, D)$, should be inserted into the queue. Note also that the definition corresponds to the informal description on scheduling strategy in Section 2. In particular, the known scheduling strategies such as EDF, SJF, and FPS all satisfy the condition defined.

3.2 Decidability and Undecidability Results

First, we consider the case of non-preemptive scheduling to introduce the problems. We have the following positive result.

Theorem 1 *The problem of checking schedulability relative to a non-preemptive scheduling strategy for task automata is decidable.*

Proof. A detailed proof is given in [2]. We sketch the proof idea here. It is to show that the schedulability question for a task automaton can be translated as a reachability question for a timed automaton.

We transform the underlying timed automaton of a task automaton $A = \langle N, l_0, E, I, M, x_{done} \rangle$ to a modified timed automaton $E(A)$ as follows. We remove all labels, and add a label $\mathsf{release}_i$ on all edges leading to a location l such that $M(l) = P_i$. This gives us a possibility to keep the information about which task is released when the automaton enters l.

We code the task queue and operations on the queue related to the given scheduling strategy as a timed automaton (called the scheduler) denoted $E(\mathsf{Sch})$. This automaton remembers the discrete parts of the queues in the locations and it uses clocks to remember the accumulated computation times and the relative deadlines for the released task instances. It is sufficient to encode only the queues from the adequate states. Therefore, there are only finitely many different discrete parts of the queues. The edges of $E(\mathsf{Sch})$ are induced by the timed automata for the scheduling strategy Sch from Definition 7 and by the rules for the task finishing from Definition 2.

The scheduler automaton manipulates the clocks as follows: Whenever an instance of a task type P_i is released by an event $\mathsf{release}_i$, a clock x^d_{ij} is reset to 0, for some j such that x^d_{ij} is not used by any other task instance. Whenever a released task instance p_{ij} is started to run, a clock x^c_{ij} is reset to 0. Whenever the value of clock x^c_{ij} of the running task instance is greater than or equal to B_i, the task instance can be removed from the queue and the next task instance can start to run. Note that this value should never be greater than W_i. Whenever the constraints $x^d_{ij} = D_i$ and $x^c_{ij} < W_i$ are met, an error state should be reached. Whenever the scheduling strategy needs to access the remaining best case/worst case computation time or the remaining deadline of a task instance (to compare it with a constant), it can use $B_i - x^c_{ij}$, $W_i - x^c_{ij}$, $D_i - x^d_{ij}$

for the running task instance ($B_i, W_i, D_i - x^d_{ij}$ for a released but not running task instance), respectively.

Finally we construct the product automaton $E(\mathsf{Sch}) \parallel E(A)$ in which both $E(\mathsf{Sch})$ and $E(A)$ can only synchronize on identical actions namely $\mathsf{release}_i$'s. It can be proved that if an error state of the product automaton is reachable, the original task automaton is non-schedulable. $\square$

For *preemptive scheduling* strategies, it has been conjectured that the schedulability checking problem is undecidable. The reason is that if we use the same ideas as for non-preemptive scheduling to encode a preemptive scheduling strategy, we must use stopwatches (or integrators) to accumulate computation times for suspended tasks. That is, it appears that the computation model behind preemptive scheduling is stopwatch automata for which it is known that the reachability problem is undecidable. However, we have positive results for the following two classes of task automata.

Theorem 2 *The problem of checking schedulability relative to a preemptive scheduling strategy is decidable for task automata with fixed computation times.*

This theorem follows from Lemma 3, 5, and 6 established in the following section.

The result holds also for task automata with interval computation times in case the finishing time of a task has no influence on the new task releases of the automata and on the decisions of the scheduling strategy (EDF and FPS are such scheduling strategies). In fact, it can be converted to the case of task automata with fixed computation times.

Theorem 3 *The problem of checking schedulability relative to* FPS *or* EDF *scheduling strategy for task automata without task feedback is decidable.*

To prove this, we show that if a non-schedulable state is reachable then it is reachable also when the computations of all tasks take the worst-case execution time. This is formulated as Lemma 7 and proved in the following section. This fact together with Theorem 2 proves Theorem 3.

From scheduling theory [1], we know that the preemptive version of the earliest deadline first (EDF) scheduling strategy is optimal in the sense that if a task queue is non-schedulable with EDF, it cannot be schedulable with any other scheduling strategy (preemptive or non-preemptive). Thus, the general schedulability checking problem is equivalent to the relative schedulability checking with respect to EDF.

For decidability, we have a general result that follows from the above theorems.

Theorem 4 *The problem of checking schedulability is decidable for task automata without task feedback or with fixed computation times.*

Unfortunately the schedulability problem is undecidable for preemptive scheduling when tasks may have variable computation times and the finishing time of a task may be used to influence (i.e., feedback on) the behaviour of the automaton. This is stated in the following theorem which is our second main technical result.

Theorem 5 *The problem of checking whether a task automaton is schedulable with* FPS *is undecidable.*

This result is established in Section 5. However, the proof does not depend on fixed priority scheduling strategy and it can be easily modified for almost all preemptive scheduling strategies (e.g., the proof holds for EDF and SJF without any modification).

4 Decidability

We shall encode the schedulability checking problem for task automata with fixed computation time of tasks as a reachability problem of timed automata with bounded subtraction. We show that a reachability problem for this extension of timed automata is decidable. In the last part of this section, we show that the schedulability checking problem for task automata without task feedback can be reduced to the schedulability checking of task automata with fixed computation time of tasks.

4.1 Timed Automata with Subtraction

We shall identify a class of suspension automata [3] that are timed automata with subtraction in which clocks may be updated by subtraction under certain conditions. We show that if for each clock there is a known maximal constant such that subtraction operations are performed on clocks only in the bounded zone, the reachability problem is decidable. Because the schedulability checking problem can be coded as a reachability problem for such automata, it is decidable.

Definition 8 *A timed automaton with subtraction is a timed automaton in which clocks may be updated by subtraction in the form $x := x - C$ in addition to reset of the form $x := 0$, where C is a natural number.*

This is the class of so-called suspension automata [3], for which it is known that the reachability problem is undecidable. However, for the following class of suspension automata, the location reachability problem is decidable.

Definition 9 *(Timed Automata with Bounded Subtraction) A timed automaton with* bounded subtraction *is a timed automaton such that there is a constant M_x for each clock x* (the ceiling of x), *and for all its reachable states (l, u)*

(1) $u(x) \geq 0$ for all clocks x, i.e., clock values should not be negative and
(2) $u(x) \leq M_x$ if $l \xrightarrow{g\, a\, r} l'$ for some l' and C such that $u \models g$ and $(x := x - C) \in r$.

Note that the two conditions imply that in a state (l, u), a clock x is allowed to be subtracted by a constant C only if $C \leq u(x) \leq M_x$.

Because subtractions on clocks are performed only when clocks are bounded with known constants, they preserve the standard region equivalence [5]. It is illustrated in Figure 3.

Definition 10 *(Region Equivalence $\sim$ [5]) For a clock $x \in \mathcal{C}$, let C_x be a natural number. For a real number t, let $\{t\}$ denote the fractional part of t, and $\lfloor t \rfloor$ denote its integer part. Let $u, v \in \mathcal{V}$. We define $u \sim v$, i.e., u, v are region-equivalent iff*

(1) for each clock x, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or $u(x) > C_x$ and $v(x) > C_x$ and
(2) for all clocks x, y if $u(x) \leq C_x$ and $u(y) \leq C_y$ then
  (a) $\{u(x)\} = 0$ iff $\{v(x)\} = 0$ and
  (b) $\{u(x)\} \leq \{u(y)\}$ iff $\{v(x)\} \leq \{v(y)\}$

However, the standard region construction above deals only with automata containing no diagonal constraints, i.e., bounds on the clock differences. To encode scheduling problems, we need to use diagonal constraints as guards in automata. For example to check the schedulability of a task automaton related to SJF, we need to compare the difference between clocks to decide where to insert a new task. We need the refined version of region equivalence from [6].

Definition 11 *(Refined Region Equivalence $\approx$) Let $\mathcal{G}$ be a finite set of diagonal constraints in the form $x - y \bowtie N$ where N is a natural number. Let $u, v \in \mathcal{V}$. We define $u \approx v$ iff*

(1) $u \sim v$
(2) $u \models g$ iff $v \models g$ for all $g \in \mathcal{G}$

For a clock assignment u, let $u(x - C)$ denote the assignment where the value of x is subtracted by C, namely $u(x - C)(x) = u(x) - C$ and $u(x - C)(y) = u(y)$ for

$y \neq x$. The following congruence properties of the refined region equivalence will give rise to a finite partitioning of the reachable state space for a timed automaton with bounded subtraction.

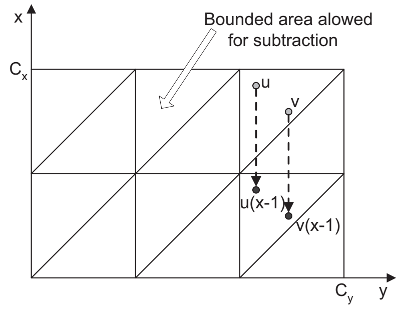

Fig. 3. Region equivalence preserved by subtraction when clocks are bounded.

Lemma 1 *Given a timed automaton with bounded subtraction, let $\mathcal{G}$ denote the set of diagonal constraints appearing in the automaton and C_x be the maximum of M_x (the ceiling of x) and all constants appearing in the guards and invariants of the automaton involving clock x. Let $u, v \in \mathcal{V}$ and t be a non-negative real number. Then $u \approx v$ implies*

(1) $u + t \approx v + t'$ for some real number t' such that $\lfloor t \rfloor = \lfloor t' \rfloor$.
(2) $u[x \mapsto 0] \approx v[x \mapsto 0]$ for a clock x,
(3) $u(x - C) \approx v(x - C)$ for all natural numbers C such that $C \leq u(x) \leq C_x$.

Proof. The proof is given in Appendix. $\square$

The refined region equivalence induces a bisimulation over reachable states of timed automata with bounded subtraction, which can be used to partition the whole state space into a finite number of equivalence classes.

Lemma 2 *Assume a timed automaton with bounded subtraction, a location l and clock assignments u and v. Then $u \approx v$ implies that*

(1) whenever $(l, u) \longrightarrow (l', u')$ then $(l, v) \longrightarrow (l', v')$ for some v' s.t. $u' \approx v'$.
(2) whenever $(l, v) \longrightarrow (l', v')$ then $(l, u) \longrightarrow (l', u')$ for some u' s.t. $u' \approx v'$.

Proof. It follows from Lemma 1. $\square$

The above lemma essentially states that if $u \approx v$ then (l, u) and (l, v) are bisimilar, which implies the following result.

Lemma 3 *The location reachability problem for timed automata with bounded subtraction is decidable if the bound M_x for each clock x is known.*

Proof. From Lemma 2, it follows that for each location l of the automaton, there is a finite number of equivalence classes induced by the bisimulation relation $\approx$. Because the number of locations of an automaton is finite, the whole state space of an automaton can be partitioned into finite number of such equivalence classes and these equivalence classes can be effectively generated. □

4.2 Preemptive Schedulers as Timed Automata with Subtraction

We shall first consider automata with fixed computation times. Later on, we show how these results extend to task automata without task feedback, that are automata with tasks whose computation times can be intervals but the completion time of a task has no influence on the new releases of task instances.

Because we consider only tasks with fixed computation times, i.e., $B = W$ for each task type $P(B, W, D)$, in the following, we will talk only about the worst case remaining computation time w_i for each task instance $P_i(b_i, w_i, d_i)$ and W_i as a parameter of the task type.

At any time point, the task queue may contain a sorted list of task instances. To be able to talk about task instances according to their task type, we assume some (any) order on the task types. By p_{ij} we denote a j-th task instance of the i-th task type. Note that both indices are bounded, one by the number of the task types and the other one by the maximal number of task instances of a particular task type in a not overflowed task queue. Indices of task instances of the same type do not correspond to their order in the queue, they just allow us to distinguish between two instances of the same type. They are also reused when a task instance finishes.

Main ideas: Essentially, the decidability results are achieved by the encoding of task preemptions using subtraction on clocks by constants (instead of using stopwatches). The difficult part is to show how does the automaton keep the information about the remaining computation time and deadline of the task instances in the queue using clocks. For each released task instance, we use two clocks: a computing clock to remember the accumulated computation time the instance has consumed so far, and a deadline clock to remember the remaining relative deadline since release. To deal with preemption, we use subtraction. When the running task is preempted, we do not stop the computing clock as we may do in using stopwatch automata. But when a running task finishes, we subtract the computing clocks (for all preempted instances) with the execution time of the finished task, which is precisely the amount of time that the computing clocks have proceeded since the most recent preemption. Assume that the current running task p_{ij} and the task p_{kl} preempted by the current one have computing clocks x^c_{ij} and x^c_{kl}, and their original fixed computation times are W_i and W_k (known constants) respectively. Then $W_i - v(x^c_{ij})$

stores precisely the remaining computation time for the running task (v is a clock valuation). The remaining computation time for the preempted task is $W_k - (v(x^c_{kl}) - v(x^c_{ij}))$. Following this reasoning, we may recover the remaining computation times using the clock differences for all preempted tasks according to the ordering they are preempted. For example, the remaining computation time for the task instance p_{mn} preempted by p_{kl} is $W_m - (v(x^c_{mn}) - v(x^c_{kl}))$. Using diagonal constraints, the remaining computation times of the preempted task instances may be compared with the known constant computation time of a newly released task instance. Thus a scheduling strategy deciding where the new task instance should be inserted in the task queue based on remaining computation times, can be encoded as a timed automaton with diagonal constraints and bounded subtraction.

Assume a task automaton A with fixed computation times, and a *preemptive scheduling* strategy Sch. As for the non-preemptive case (Theorem 1), let $E(A)$ denote the timed automaton of A where the action label of each edge is replaced with $\mathsf{release}_i$ if its target location is mapped to P_i. For a preemptive scheduling strategy Sch, we construct $E(\mathsf{Sch})$ as a timed automaton extended with subtraction to synchronize with $E(A)$ through the actions $\mathsf{release}_i$'s. This automaton encodes the queues from all possible adequate states (the task queue is not overflowed and no deadline is violated) in the locations. The edges between the locations are induced by the scheduling strategy and the rules for the task finishing.

We shall show that the LTS induced by $E(A) \parallel E(\mathsf{Sch})$ for any A and Sch is weakly timed bisimilar to $[\![A_{\mathsf{Sch}}]\!]$, i.e., the LTS induced by A with respect to Sch, restricted to the adequate states. We shall also show how to detect reachability of non-adequate states in $[\![A_{\mathsf{Sch}}]\!]$.

Each location of $E(\mathsf{Sch})$ encodes the information about the discrete part of the queue, containing the names of the types of the task instances and their status: running, preempted, or just released. We encode this information into variables (with finite domain). For each p_{ij}, we use a state variable $\mathsf{Status}(ij)$ initialized to free, representing the current status of p_{ij}:

- $\mathsf{Status}(ij) = \mathsf{free}$ means that the position i, j in the task queue is free (i.e., p_{ij} is finished or not released yet),
- $\mathsf{Status}(ij) = \mathsf{released}$ means that p_{ij} is released, not started yet,
- $\mathsf{Status}(ij) = \mathsf{running}$ means that p_{ij} is running on the processor, and
- $\mathsf{Status}(ij) = \mathsf{preempted}$ means that p_{ij} has started its computation but it is suspended now.

The remaining computation time w_{ij} of a preempted task p_{ij} ($\mathsf{Status}(ij) = \mathsf{preempted}$) may be a real number; but if p_{ij} is only released ($\mathsf{Status}(ij) = \mathsf{released}$), w_{ij} is an integer W_i. We introduce two clocks for each p_{ij}:

- x^c_{ij} (a computing clock) is used to remember the accumulated computation time since p_{ij} was started.
- x^d_{ij} (a deadline clock) is used to remember the deadline and it is reset to 0 when p_{ij} is released.

Since we cannot stop a computing clock for a preempted task instance p_{ij}, we have to remember which task instance has preempted it (or which task instance is the closest one running instead p_{ij}). The predicate $\mathsf{Preempted\_by}(ij)(kl)$ is true if and only if $\mathsf{Status}(ij) = \mathsf{preempted}$, and p_{kl} is the closest task instance in the queue to p_{ij} such that $\mathsf{Status}(kl) = \mathsf{preempted}$ or $\mathsf{Status}(kl) = \mathsf{running}$, and p_{kl} is closer to the head of the queue than p_{ij}.

We will use locations to remember which (if any) task instance is running at the moment. $E(\mathsf{Sch})$ has the following types of locations:

- Idling denotes that the task queue is currently empty, that is, $\mathsf{Status}(ij) = \mathsf{free}$ for all i, j.
- $\mathsf{Running}(ij)$ denotes that a task instance p_{ij} is running, that is, $\mathsf{Status}(ij) = \mathsf{running}$. Each such location has invariants $x^c_{ij} \leq W_i$ and $x^d_{kl} \leq D_k$ for all k, l such that $\mathsf{Status}(kl) \neq \mathsf{free}$.

We also need to keep track of the order of the task instances in $E(\mathsf{Sch})$. For each p_{ij}, we use a state variable $\mathsf{Position}(ij)$ which is set to the value $\perp$ for all i, j such that $\mathsf{Status}(ij) = \mathsf{free}$ and it is set to a natural number denoting the position of p_{ij} in the queue otherwise. If $\mathsf{Status}(ij) = \mathsf{running}$ then $\mathsf{Position}(ij) = 1$. Since the queue size is bounded in all adequate states, $\mathsf{Position}(ij)$ has a finite domain for all p_{ij} (and thus can be encoded into timed automata).

When a task finishes or a new one arrives, we use a predicate $\mathsf{Head}(mn)$ to denote the fact that the task instance p_{mn} is the next task instance to be executed (the next head of the queue). If a new task instance arrives, we take it also into account. Conversely, when a task finishes, we do not consider it anymore.

We need to update the positions of the tasks in the queue when a task instance arrives or finishes. Let $\mathsf{Remove}(mn)$ denote the update of the variables $\mathsf{Position}(ij)$ for all p_{ij} when the task instance p_{mn} is removed from the queue (it has finished). When a new instance p_{mn} arrives, $\mathsf{Insert}(mn)$ denotes the update of the variables $\mathsf{Position}(ij)$ corresponding to inserting the new task instance into the (discrete part of the) queue according to the scheduling strategy.

According to our definition of scheduling strategies, $\mathsf{Head}(mn)$, $\mathsf{Remove}(mn)$, and $\mathsf{Insert}(mn)$ can be encoded using (diagonal free) timed automata guards on the values w_{ij}, d_{ij} of the task instances. These values can be at any timepoint reconstructed from the clocks x^c_{ij} and x^d_{ij} (as is proved below) and therefore the predicates can be encoded as (diagonal) guards on the edges of $E(\mathsf{Sch})$. We show how this can be done for EDF and SJF in the Appendix (the case of FPS is straightforward, because it uses only the static task parameters).

The edges of $E(\mathsf{Sch})$ between its locations are defined in Table 1. Later we also add a location Err which will be reachable if and only if either a deadline can be missed or the queue can overflow.

(1) Idling to Running(ij):
- guard: none
- action: $\mathsf{release}_i$
- reset: $x^c_{ij} := 0$, $x^d_{ij} := 0$, $\mathsf{Status}(ij) := \mathsf{running}$

(2) Running(ij) to Idling:
- guard: $\mathsf{Status}(kl) = \mathsf{free}$ for all $k, l \neq i, j$, $x^c_{ij} = W_i$
- action: none
- reset: $\mathsf{Status}(ij) := \mathsf{free}$, $x_{done} := 0$, $\mathsf{Remove}(ij)$

(3) Running(ij) to Running(mn): there are three types of edges.
  (a) The running task instance p_{ij} is finished, p_{mn} was preempted and is scheduled to run now:
  - guard: $x^c_{ij} = W_i$, $\mathsf{Status}(mn) = \mathsf{preempted}$, $\mathsf{Head}(mn)$
  - action: none
  - reset: $\mathsf{Status}(ij) := \mathsf{free}$, $x_{done} := 0$, $x^c_{kl} := x^c_{kl} - W_i$ for all k, l s.t. $\mathsf{Status}(kl) = \mathsf{preempted}$, $\mathsf{Status}(mn) := \mathsf{running}$, $\mathsf{Remove}(ij)$

  (b) A new task instance p_{mn} is released, which preempts the running task instance p_{ij}:
  - guard: $\mathsf{Status}(mn) = \mathsf{free}$, $\mathsf{Head}(mn)$
  - action: $\mathsf{release}_m$
  - reset: $\mathsf{Status}(mn) := \mathsf{running}$, $x^c_{mn} := 0$, $x^d_{mn} := 0$, $\mathsf{Status}(ij) := \mathsf{preempted}$, $\mathsf{Insert}(mn)$

  (c) The running task instance p_{ij} is finished and p_{mn} (which was released but has never run) is scheduled to run now:
  - guard: $x^c_{ij} = W_i$, $\mathsf{Status}(mn) = \mathsf{released}$, $\mathsf{Head}(mn)$
  - action: none
  - reset: $\mathsf{Status}(ij) := \mathsf{free}$, $x_{done} := 0$, $x^c_{mn} := 0$, $x^c_{kl} := x^c_{kl} - W_i$ for all k, l s.t. $\mathsf{Status}(kl) = \mathsf{preempted}$, $\mathsf{Status}(mn) := \mathsf{running}$, $\mathsf{Remove}(ij)$

(4) Running(ij) to Running(ij): a new task is released and the running task instance p_{ij} will continue to run:
- guard: $\mathsf{Status}(mn) = \mathsf{free}$, $\mathsf{Head}(ij)$
- action: $\mathsf{release}_m$
- reset: $\mathsf{Status}(mn) := \mathsf{released}$, $x^d_{mn} := 0$, $\mathsf{Insert}(mn)$

Table 1
Defining the edges of $E(\mathsf{Sch})$.

At first, we prove that $E(A) \parallel E(\mathsf{Sch})$ is a faithful translation of A and

Sch restricted to the adequate states by showing a weak timed bisimulation between their semantical labeled transition systems.

Lemma 4 *The LTS induced by $E(A) \parallel E(\mathsf{Sch})$ is weak timed bisimilar to $[\![A_{\mathsf{Sch}}]\!]$ restricted to the adequate states, based on an abstract transition relation which abstracts away from action labels, defined as follows:*

(1) $\mathsf{s}_A \xrightarrow{\tau} \mathsf{s}'_A$ *iff* $\mathsf{s}_A \xrightarrow{a} \mathsf{s}'_A$ *for an action* $a \in \mathcal{A}ct \cup \{fin\}$.
(2) $\mathsf{s}_A \xrightarrow{t} \mathsf{s}'_A$ *iff* $\mathsf{s}_A \xrightarrow{t} \mathsf{s}'_A$ *for a time delay* t.

where $\mathsf{s}_A, \mathsf{s}'_A$ *are both either states of* $[\![A_{\mathsf{Sch}}]\!]$ *or states of* $E(A) \parallel E(\mathsf{Sch})$.

PROOF.

Assume that the task queue in a state (l, u, q) of A, contains triples (b_{ij}, w_{ij}, d_{ij}) where $b_{ij} = w_{ij} \geq 0$ and d_{ij} are the remaining computation time and relative deadline for a task instance p_{ij} (we use the same indexing as in $E(\mathsf{Sch})$). By $p_{ij} <_q p_{kl}$ we denote the fact that $p_{ij} \in q$, $p_{kl} \in q$, and the task instance p_{ij} in q is closer to the head of the queue than the task instance p_{kl}.

We define the following relations:

$S_1 = \{((l, u, []), (\langle l, \mathsf{Idling}\rangle, (u \cup v)))|l \in N, u, v \in \mathcal{V}\}$
$S_2 = \{((l, u, q), (\langle l, \mathsf{Running}(mn)\rangle, (u \cup v))) \mid l \in N, u, v \in V \; C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5\}$ where

- $C_1 \equiv [p_{ij} \in q \Leftrightarrow \mathsf{Status}(ij) \neq \mathsf{free} \wedge p_{ij} <_q p_{kl} \Leftrightarrow \mathsf{Position}(ij) < \mathsf{Position}(kl)]$
- $C_2 \equiv [w_{ij} = W_i - v(x^c_{ij})$ for i, j s.t. $\mathsf{Status}(ij) = \mathsf{running}]$
- $C_3 \equiv [w_{ij} = W_i - (v(x^c_{ij}) - v(x^c_{kl}))$ for all i, j s.t. $\mathsf{Status}(ij) = \mathsf{preempted}$ and k, l s.t. $\mathsf{Preempted\_by}(ij)(kl)]$
- $C_4 \equiv [w_{ij} = W_i$ for all i, j s.t. $\mathsf{Status}(ij) = \mathsf{released}]$
- $C_5 \equiv [d_{ij} = D_i - v(x^d_{ij})$ for all i, j s.t. $\mathsf{Status}(ij) \neq \mathsf{free}]$

We establish that $S = S_1 \cup S_2$ is a weak timed bisimulation.

First consider a pair in S_1, $((l, u, []), (\langle l, \mathsf{Idling}\rangle, (u \cup v)))$.

- If A or $E(A)$ takes a discrete transition which does not release a task instance (is not labeled by a $\mathsf{release}_i$) or a time delay then the other automaton can simply take the same transition. Note, that $E(\mathsf{Sch})$ can delay in Idling for an unbounded amount of time.
- If A takes a discrete transition releasing an instance of P_i then $E(\mathsf{Sch})$ takes the corresponding transition labeled by $\mathsf{release}_i$ and moves to $\mathsf{Running}(ij)$ (Rule 1 in Table 1). It is easy to check that conditions C_1–C_5 hold, because x^c_{ij} and x^d_{ij} are reset and $w_{ij} = W_i, d_{ij} = D_i$.
- If $E(\mathsf{Sch})$ takes a transition labeled by $\mathsf{release}_i$ and moves to $\mathsf{Running}(ij)$

(Rule 1 in Table 1) then A takes the corresponding discrete transition which releases an instance of P_i.

Now we consider pairs in S_2 and possible transitions according to their type.

Discrete transition. Assume $((l, u, q), (\langle l, \mathsf{Running}(ij)\rangle, (u \cup v))) \in S_2$ and A takes a transition $(l, u, q) \longrightarrow (l', u', \mathsf{Sch}(M(l'), q))$. Further assume that this transition is induced by $l \xrightarrow{g\,a\,r} l'$, $u \models g$, and $u[r] \models I(l')$. Then $E(A) \parallel E(\mathsf{Sch})$ takes the corresponding transition. There are three possibilities:

- $M(l') = \bot$ — the corresponding transition in $E(A) \parallel E(\mathsf{Sch})$ is enabled and both resulting states trivially belong to S_2.
- $M(l') = P_m(B_m, W_m, D_m)$ and the new instance preempts the currently running one, i.e., $\mathsf{Sch}(P_m(B_m, W_m, D_m), q) = P_m(B_m, W_m, D_m) :: q$ — the corresponding transition in $E(A) \parallel E(\mathsf{Sch})$ (Rule 3b in Table 1) is enabled because there exists n such that $\mathsf{Status}(mn) = \mathsf{free}$ (A is restricted to not overflowed queues) and $\mathsf{Head}(mn)$ is true. Conditions C_1–C_5 hold.
- $M(l') = P_m(B_m, W_m, D_m)$ and $\mathsf{Sch}(P_m(B_m, W_m, D_m), P_{ij}(b_{ij}, w_{ij}, d_{ij}) :: q) = P_{ij}(b_{ij}, w_{ij}, d_{ij}) :: q'$ — the corresponding transition in $E(A) \parallel E(\mathsf{Sch})$ (Rule 4 in Table 1) is enabled because there exists n such that $\mathsf{Status}(mn) = \mathsf{free}$ (A is restricted to not overflowed queues) and $\mathsf{Head}(ij)$ is true. Conditions C_1–C_5 hold in the new pair of states.

The other direction is symmetrical if we notice that $\mathsf{Head}(kl)$ can be true only for either i, j or m, n (and for only one of them).

Time pass. Assume $((l, u, P_{ij}(b_{ij}, w_{ij}, d_{ij}) :: q), (\langle l, \mathsf{Running}(ij)\rangle, (u \cup v))) \in S_2$ and A takes a transition $(l, u, P_{ij}(b_{ij}, w_{ij}, d_{ij}) :: q) \xrightarrow{t} (l, u', \mathsf{Run}(P_{ij}(b_{ij}, w_{ij}, d_{ij}) :: q, t))$. We know that $t \leq w_{ij}$ and $u + t \models I(l)$. Then $E(A) \parallel E(\mathsf{Sch})$ delays for the same amount of time. $E(A)$ can delay for t, $u + t \models I(l)$ is a sufficient condition. For $E(\mathsf{Sch})$, we need to satisfy $(v + t)(x^c_{ij}) \leq W_{ij}$ and $(v + t)(x^d_{kl}) \leq D_{kl}$ for all k, l such that $\mathsf{Status}(kl) \neq \mathsf{free}$. The former holds because of Condition C_2, the latter because of C_5 and the fact that A is restricted to the states where no deadline is missed. Conditions C_1–C_5 hold in the new pair of states.

The other direction is symmetrical, Conditions C_1–C_5 are formulated as equalities.

Finishing a task. Assume $((l, u, P_{ij}(b_{ij}, w_{ij}, d_{ij}) :: q), (\langle l, \mathsf{Running}(ij)\rangle, (u \cup v))) \in S_2$ and A takes a transition $(l, u, P_{ij}(b_{ij}, w_{ij}, d_{ij}) :: q) \xrightarrow{\tau} (l, u[x_{done}], q)$. We know that $w_{ij} = 0$ and $u[done] \models I(l)$. Because of Condition C_2, we know that $x^c_{ij} = W_i$. If $q = []$ then the transition given by Rule 2 in Table 1 is enabled and $E(A) \parallel E(\mathsf{Sch})$ takes it. Conditions C_1–C_5 hold. Otherwise, $\mathsf{Head}(mn)$ is true for some (unique) m, n.

If $\mathsf{Status}(mn) = \mathsf{preempted}$ then $E(A) \parallel E(\mathsf{Sch})$ takes the transition given by Rule 3a in Table 1. Subtraction of clocks in the reset together with Condition C_3 ensure that C_2 and C_3 hold in the new pair of states. The other conditions also hold.

If $\mathsf{Status}(mn) = \mathsf{released}$ then $E(A) \parallel E(\mathsf{Sch})$ takes the transition given by Rule 3c in Table 1. Subtraction of clocks in the reset together with Condition C_3 ensure that C_3 hold in the new pair of states. The other conditions also hold.

Again, the other direction is symmetrical. $\square$

Now we show that the product automaton $E(A) \parallel E(\mathsf{Sch})$ is a bounded timed automaton with subtraction.

Lemma 5 *The automaton $E(A) \parallel E(\mathsf{Sch})$ for a task automaton A and a preemptive scheduling strategy Sch is a timed automaton with bounded subtraction.*

Proof. Note that $x^c_{kl} \leq x^d_{kl}$ because whenever x^d_{kl} is reset to zero, so is x^c_{kl} (when a new instance of P_k is released). Note also that all edges labeled with a subtraction lead from and to a location with the invariant $x^d_{kl} \leq D_k$ for all k, l such that $\mathsf{Status}(kl) \neq \mathsf{free}$. Thus x^d_{kl} is bounded by D_k and therefore x^c_{kl} is bounded by D_k for k, l such that $\mathsf{Status}(kl) \neq \mathsf{free}$. But only such clocks can be subtracted in $E(\mathsf{Sch})$.

Secondly, the only possibility for a computing clock, say x^c_{kl} for a task instance p_{kl}, to become negative is by a subtraction. But a subtraction is done on x^c_{kl} only when a task instance, say p_{ij}, is finished, i.e., $x^c_{ij} = W_i$ holds. Note that $\mathsf{Status}(kl) = \mathsf{preempted}$ implies that p_{kl} was released and started (when $\mathsf{Status}(kl)$ was set to $\mathsf{running}$) before $\mathsf{Status}(ij)$ was set to $\mathsf{running}$. Otherwise, $\mathsf{Status}(kl) = \mathsf{released}$. That is x^c_{kl} is reset to zero before x^c_{ij}. Thus we have $x^c_{kl} \geq x^c_{ij}$ implying that $x^c_{kl} - W_i \geq 0$ when $x^c_{ij} = W_i$. Therefore, all clocks are non-negative. $\square$

To detect whether a non-adequate state is reachable in $[\![A_{\mathsf{Sch}}]\!]$, we add a location Err and the edges to $E(\mathsf{Sch})$ as described in Table 2.

Now we have the correctness lemma for our encoding. Let v_0 be a clock valuation which assigns zero to all clocks of the scheduler automaton $E(\mathsf{Sch})$.

Lemma 6 *Let A be a task automaton with fixed computation time of tasks and Sch a preemptive scheduling strategy. Assume that $(l_0, u_0, [])$ and $(\langle l_0, \mathsf{Idling}\rangle, u_0 \cup v_0)$ are the initial states of A and the product automaton $E(A) \parallel E(\mathsf{Sch})$ respectively. Then*

(1) $\mathsf{Running}(ij)$ to Err:

- guard: $x^c_{ij} < W_i$, $x^d_{ij} = D_i$
- action: none
- reset: none

(2) $\mathsf{Running}(ij)$ to Err: for each k, l there is an edge labeled with

- guard: $\mathsf{Status}(kl) = \mathsf{released}$, $x^d_{kl} > D_k - W_k$
- action: none
- reset: none

(3) $\mathsf{Running}(ij)$ to Err: for each k, l there is an edge labeled with

- guard: $\mathsf{Status}(kl) = \mathsf{preempted}$, $x^c_{ij} < W_i$, $x^d_{kl} = D_k$
- action: none
- reset: none

(4) $\mathsf{Running}(ij)$ to Err: for each k there is an edge labeled with

- guard: $\mathsf{Status}(kl) \neq \mathsf{free}$ for all l
- action: $\mathsf{release}_k$
- reset: none

Table 2
Defining the edges of $E(\mathsf{Sch})$ leading to Err.

(1) if there are l and u such that $(l_0, u_0, []) \longrightarrow^* (l, u, q_{err})$ then $(\langle l_0, \mathsf{Idling}\rangle, u_0 \cup v_0) \longrightarrow^* (\langle l', \mathsf{Err}\rangle, u' \cup v)$ for some l', u', and v, and
(2) if there are l, u, and v such that $(\langle l_0, \mathsf{Idling}\rangle, u_0 \cup v_0) \longrightarrow^* (\langle l, \mathsf{Err}\rangle, u \cup v)$ then $(l_0, u_0, []) \longrightarrow^* (l', u', q_{err})$ for some l', u'.

Proof. Consider a finite path ρ in $[\![A_{\mathsf{Sch}}]\!]$ leading to an error state (l, u, q_{err}). Let (l_e, u_e, q_e) denote the last state on the prefix of ρ containing only adequate states.

If a deadline will be missed in the immediate successor of (l_e, u_e, q_e) then there must be a task instance $P_{kl}(b_{kl}, w_{kl}, d_{kl}) \in q_e$ such that $d_{kl} = 0$ and $w_{kl} > 0$. According to Lemma 4, a corresponding state in $E(A) \parallel E(\mathsf{Sch})$ is reachable. Because of Conditions C_1–C_5, one of the transitions given by Rules 1–3 in Table 2 is enabled according to the status of $P_{kl}(b_{kl}, w_{kl}, d_{kl})$. If the queue will overflow in the immediate successor of (l_e, u_e, q_e) then a transition given by Rule 4 in Table 2 is enabled because of C_1.

If an error state is reachable in the LTS induced by $E(A) \parallel E(\mathsf{Sch})$ then it is either by taking a transition given by Rules 1–3 in Table 2 or by the transition given by Rule 4 in Table 2. In the former case, there must be a task instance $P_{kl}(b_{kl}, w_{kl}, d_{kl})$ such that $d_{kl} = 0$ and $w_{kl} > 0$ in the task queue according to Lemma 4. In the latter case, the queue overflows. Because the time does not stop in A, a deadline will be missed in both cases. $\square$

The above lemma states that the schedulability analysis problem can be solved by reachability analysis for timed automata extended with subtraction. From Lemma 5, we know that $E(\mathsf{Sch})$ is bounded. Because the reachability problem is decidable due to Lemma 3, we complete the proof for our main result stated in Theorem 2.

4.3 Task Automata without Task Feedback

Now we prove that a task automaton without task feedback is schedulable with a preemptive scheduling strategy which is either FPS or EDF if and only if it is schedulable (with the same scheduling strategy) when the tasks have constant computation times equal to the given worst-case computation times.

Lemma 7 *Let A be a task automaton without task feedback with an initial state $(l_0, u_0, [])$ and* Sch *be either* FPS *or* EDF*. If there is a path $r_1 = (l_0, u_0, []) \longrightarrow^* (l, u, q_{err})$ in $[\![A_{\mathsf{Sch}}]\!]$ then there is a path $r_2 = (l_0, u_0, []) \longrightarrow^* (l', u', q_{err})$ in $[\![A_{\mathsf{Sch}}]\!]$ such that each task instance $P(b, w, d)$ finishes when $w = 0$.*

Proof. Given a path r_1, we construct r_2 so that we let the underlying timed automaton of A to perform the same transitions as in r_1 (discrete transitions are taken at the same absolute time points) and all task instances $P(b, w, d)$ finish when $w = 0$ (task finishing transitions are taken at possibly different absolute time points). We have to show that r_2 is a path in $[\![A_{\mathsf{Sch}}]\!]$ and that it leads to a state where a deadline is missed.

Because A has no task feedback, the differences in the absolute time points at which task finishing transitions are taken cannot influence the underlying timed automaton of A. In particular, validity of the invariants in the locations is not influenced and all transitions that were enabled along r_1 are also enabled along r_2 (at the same time points). Also, task instances cannot be forced to finish before $w = 0$. Therefore, r_2 is a path in $[\![A_{\mathsf{Sch}}]\!]$.

To show that r_2 leads to a state where a deadline is missed we show the following proposition true. Let (l, u, q_1) and (l, u, q_2) denote the states of A at the same absolute time point along r_1 and r_2, respectively. Then q_1 can be obtained from q_2 by dropping some task instances and possibly decreasing some remaining computation times (all remaining deadlines are the same for the task instances present in both queues).

The proof is done by induction on the length of the path. The proposition trivially holds for a path of zero length.

Assume that the proposition holds in states (l, u, q_1) and (l, u, q_2). If A takes a discrete transition not releasing any task then the proposition holds trivially. If A takes a discrete transition which releases a task instance $P_{ij}(B_{ij}, W_{ij}, D_{ij})$ then we claim that the scheduling strategy inserts this task into the same position in q_1 and q_2 with respect to the tasks that are in both queues. This clearly holds for FPS, because FPS decides only according to the discrete part of the queue. EDF decides just according to the ordering of the remaining deadlines which depend only on the release times of task instances. But release times are the same for both r_1 and r_2. In other words, if a task instance $Q(b, w, d)$ has the earliest deadline in q_2 and it is still in q_1 then it also has the earliest deadline there. Thus, the proposition holds after releasing $P_{ij}(B_{ij}, W_{ij}, D_{ij})$.

If A delays for a non-zero time then the remaining computation times of both running task instances in q_1 and q_2 is decreased by the same amount. This cannot invalidate the property if the running task instances are the same. Otherwise, according to the induction hypothesis there is a corresponding task instance $Q(b, w, d)$ in q_2 to the running task instance $Q(b', w', d)$ in r_1. Then $w > w'$ after the time pass transition.

If A finishes a task instance in r_1 then the proposition holds. If A finishes a task instance $P(b, w, d)$ in r_2 then we know that $w = 0$ and from induction hypothesis either the corresponding task instance $P(b', w', d)$ in r_1 has already finished or $w' = w$ and therefore it must finish at the same time point. $\square$

As a consequence, we may consider only the worst case computation time of all tasks (all best case computation times are equal to the worst case computation times). That is, the best case execution times have no effect on the schedulability of systems without task feedback. Therefore, Lemma 7 and Theorem 2 prove Theorem 3. Note that Lemma 7 does not hold for all scheduling strategies. For example, it does not hold for SJF.

5 Undecidability

In this section we show that the schedulability problem for task automata in general (i.e., with preemption, task feedback, and variable execution times of tasks) with fixed priority scheduling is undecidable.

The proof is done by reduction of the halting problem for two-counter machine to the schedulability problem for task automata. A *two-counter machine* consists of a finite state control unit and two unbounded non-negative integer counters. Initially, both counters contain the value 0. Such a machine can execute three types of instructions: incrementation of a counter, decrementation of a counter, and branching based upon whether a specific counter contains the value 0. Note that decrementation of a counter with the value 0 leaves this counter unchanged. After execution of an instruction, a machine changes deterministically its state. One state of a two-counter machine is distinguished as *halt state*. A machine halts if and only if it reaches this state.

We present an encoding of a two-counter machine M using a task automaton A_M such that M halts if and only if A_M is non-schedulable, based on the undecidability proofs of [7]. In the construction, the states of M correspond to specific locations of A_M and each counter is encoded by a clock. We show how to simulate the two-counter machine operations. First, we adopt the notion of wrapping of [7].

Definition 12 *A task automaton over set of clocks $\mathcal{C}$ is* N-wrapping *if for all states (l, u, q) reachable from its initial state and for all clocks $x \in \mathcal{C}$: $u \models x \leq N$. An* N-wrapping edge *for a clock x and a location l is an edge from l to itself that is labeled with the guard $x = N$ and which resets the clock x. A clock that is reset only by wrapping edges is called* system clock.[5] *Each time period between two consecutive time points at which any system clock contains value* 0 *is called* N-wrapping period.

We use wrapping to simulate discrete steps of a two-counter machine. Each step is modeled by several N-wrapping periods. We define the wrapping-value of a clock to be the value of the clock when the system clock is 0. Note that a clock is carrying the same wrapping value if it is not reset by another edge than the wrapping edges. This principle is shown in Figure 4, where x_{sys} is a system clock and clock x_{copy} contains the same wrapping-value when the automaton takes transitions e_1 and e_3.

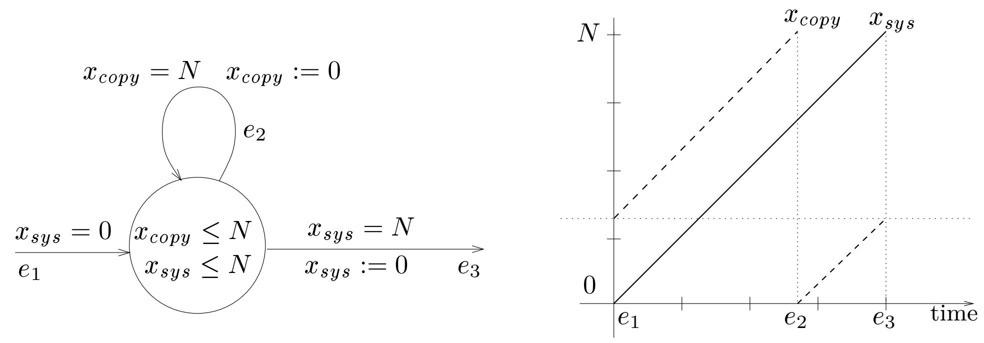

Fig. 4. The wrapping edge e_2 makes clock x_{copy} carry the same wrapping-value when the transitions e_1 and e_3 are taken.

We encode a two-counter machine M with counters C and D using a 4-wrapping automaton A_M with one system clock denoted x_{sys} and five other clocks $x_c, x_d, x_{old}, x_{copy}$, and x_{done}. In particular, we encode counters C and D of M by clocks x_c and x_d like this: counter value v corresponds to the clock

[5] Note that all system clocks contain the same value.

wrapping-value 2^{1-v}. We use the density of the continuous domain to encode arbitrarily large values of the counters. Decrementation (incrementation) of a counter corresponds to doubling (halving) the wrapping-value of the corresponding clock. Test for zero corresponds to the check whether the clock wrapping-value equals to 2.

Now we show how to simulate the decrementation operation by doubling the wrapping-value of the clock x_d. To do this, we use two tasks: P and Q. The task P has execution time in $[0, 1]$ and deadline 50; the task Q has execution time in $[8, 8]$ and deadline 100. Moreover, the priority of P is higher than the priority of Q, i.e., P always preempts Q. Notice that the execution time of task P can vary and the execution time of the task Q is fixed.

The basic idea of doubling a wrapping-value $v \in (0, 1]$ of clock x_d is as follows: we assume that the current wrapping-value of x_d is v. We copy it to the clock x_{copy} (that is, to make the wrapping-value of the clock x_{copy} to be v). Then we release the task Q non-deterministically and reset x_d. The idea is to start Q $2v$ time units before the reset of the system clock x_{sys} and to use x_d to record the response time of Q. Two instances of P are released before Q finishes, that is P preempts Q twice. We make sure that the execution time of each of these two instances of P is exactly v time units. Note that v can be any real number within the interval $(0, 1]$. Then the response time for Q is exactly $8 + 2v$. If Q finishes at a time point when the system clock x_{sys} is reset to 0, the wrapping-value of x_d is $2v$. As Q is released non-deterministically, it is enough if there is one such computation.

To simplify the presentation, we construct A_M with timestops. But it is easy to see that we could add an unguarded transition into a sink location outgoing from every location. The sink location does not release any task, it does not have any invariant and it does not have any outgoing transition. A computation leading into this location does not correspond to a computation of the two-counter machine and it does not lead into an error state (a deadline miss). Therefore, it does not influence the correctness of the reduction.

In Figure 5, we show the part of A_M that doubles the wrapping-value of the clock x_d. Figure 6 illustrates the time chart of the doubling process. Assume that a two-counter machine M is currently in a state s_i and that it wants to decrease the counter D and then move to a state s_j. The locations l_i and l_j of A_M correspond to the states s_i and s_j respectively. Note that the dashed edge shows the transition of the two-counter machine (it is not a transition of A_M). Note also that the decrementation operation leaves a counter with value 0 unchanged; the automaton can move from l_i directly to l_j through the transition e_0 when x_d contains the wrapping-value 2 (which corresponds to the counter value 0). Otherwise, the following steps are taken to double the wrapping-value of x_d.

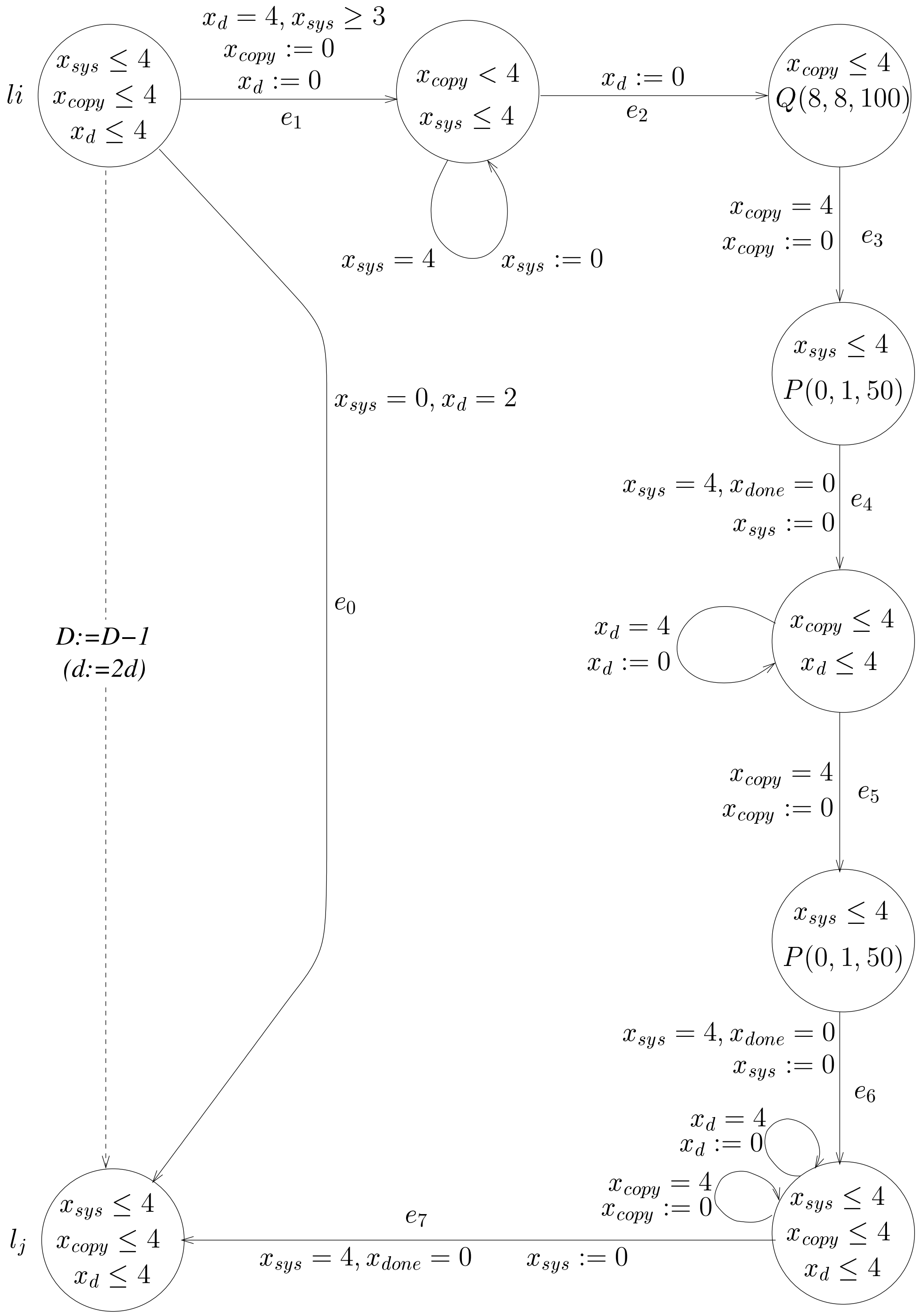

Fig. 5. A part of reduction automaton corresponding to a decrementation of D. The wrapping edges for clocks x_c, x_{old}, x_{done}, and for all clocks in locations l_i, l_j are omitted. The location invariants $x_c \leq 4, x_{old} \leq 4$, and $x_{done} \leq 4$ are also omitted as well as transitions preventing timestops.

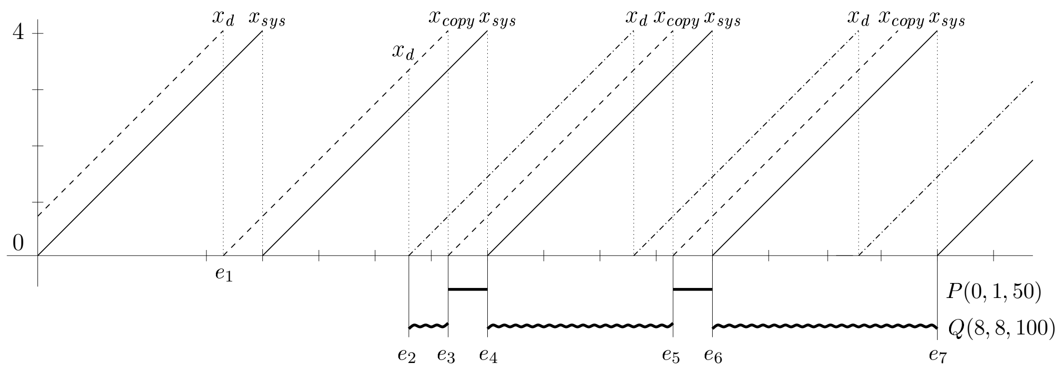

Fig. 6. Time chart of the doubling procedure.

First, the wrapping-value of x_d is copied to the clock x_{copy} (by transition e_1), that is, x_{copy} carries the same wrapping-value as x_d. Then the automaton non-deterministically guesses the doubled wrapping-value of x_d (note that when x_d is reset, it will carry a new wrapping-value). It resets x_d at a non-deterministically chosen time instant and at the same time it releases the task Q (transition e_2).

The automaton waits until the clock x_{copy} reaches time 4, then resets x_{copy} and releases P (transition e_3), which preempts Q. Note that the wrapping-value of x_{copy} will remain to be v and at this time point the value of the system clock x_{sys} is $4 - v$. Therefore, x_{sys} will reach 4 in v time units.

The next transition e_4 is guarded by two constraints: $x_{sys} = 4, x_{done} = 0$. To satisfy these constraints, the automaton has to wait in this location for v time units, and the task P must finish at this time point, which resets the clock x_{done}.[6] By this we make P run (and prevent Q from running) exactly for v time units. Now we repeat this procedure again. That is, the automaton waits until $x_{copy} = 4$. Then it releases the task P and forces it to run exactly for v time units (transitions e_5 and e_6).

Now, if the non-deterministic guess of the doubled wrapping-value of x_d was correct, task Q must finish when x_{sys} is equal to 4, which makes the guard on e_7 become true and the automaton moves to location l_j. Thus, if the location l_j is reachable, the wrapping-value of x_d is $2v$. This is stated in the following lemma.

Lemma 8 *Let (l_i, u, q) be an arbitrary state of the automaton shown in Figure 5 where $u(x_d) = v$, $v \in (0, 1]$, and $q = []$. Then (l_j, u', q') is reachable for some u' and q', and if (l_j, u', q') is reachable, it must be the case that $q' = []$, and $u'(x_d) = 2v$.*

[6] We have to make sure that x_{done} is not reset by a wrapping edge when it is tested by a guard of the automaton. This causes no technical difficulties and it is omitted from Figure 5.

PROOF. The proof is obvious from the construction in Figure 5. □

To increment a counter we need to halve a wrapping-value of a clock, say x_c. For this, we use the clock x_{old} to copy the wrapping-value of x_c. The new wrapping-value v of x_c is non-deterministically guessed and it is checked by the above doubling procedure. If the wrapping-value of x_{old} (the original wrapping-value of x_c) is $2v$, then the automaton can proceed to the location corresponding to the destination state in an increment instruction.

To simulate branching, we construct two transitions outgoing from a location with guards $x_{sys} = 0 \wedge x_c = 2$ and $x_{sys} = 0 \wedge x_c \neq 2$. The initial state of M corresponds to a location where both x_c and x_d contain the wrapping-value 2. This can be achieved by integer guards and resets.

The halt state corresponds to the location *halt* with unguarded self-loop releasing the task Q whenever it is visited. It follows that the automaton A_M is schedulable if and only if the location *halt* is unreachable, i.e., the two-counter machine M does not halt.

6 Conclusions and Related Work

We have developed a theory of *task automata*, an extended version of timed automata with asynchronous processes, i.e., computation tasks triggered by timed events, which may serve as a model for real time systems with non-uniformly recurring tasks. The model can be used to specify resource requirements and hard timing constraints on computations, in addition to features offered by timed automata. It is general and expressive enough to describe concurrency, synchronization, and tasks which may be periodic, sporadic, preemptive, and (or) non-preemptive as well as dependent relations between tasks. The classical notion of schedulability is naturally extended to the model of task automata.

Our main technical contributions include the proof that the schedulability checking problem related to preemptive scheduling is decidable for a large class of task automata. The problem has been suspected to be undecidable due to the nature of preemptive scheduling. To our knowledge, this is the first decidability result for preemptive scheduling in dense time models. We believe that our work is one step forward to bridge scheduling theory and automata-theoretic approaches to system modeling and analysis.

The negative result on task automata is that the schedulability checking problem is undecidable for the class of automata with tasks whose computation times are intervals and the completion time of a task may influence the new task releases. We have studied the borderline between decidable and undecidable cases. It is shown that the schedulability problem for many scheduling strategies will be undecidable if the following three conditions hold at the same time: (1) the execution times of tasks are intervals, (2) the precise finishing time of a task may influence the new task releases, and (3) a task is allowed to preempt another running task.

A challenge is to make the results an applicable technique combined with classical methods such as rate monotonic scheduling. We need new algorithms and data structures to represent and manipulate the dynamic task queue consisting of time and resource constraints. As another direction of future work, we shall study the schedule synthesis problem. More precisely given an automaton, it is desirable to characterize the set of schedulable traces accepted by the automaton.

Related work.

This paper summarises and extends our previous results on solving scheduling problems using timed automata; it is the full and extended version of two conference papers. The decidability results have been presented in [8] and the undecidability results in [9]. The general encoding scheme of scheduling strategies has been implemented in the TIMES tool [10], based on the UPPAAL DBM library extended with a subtraction operation. A more recent result shows that for systems where tasks are assigned fixed priorities, the schedulability analysis problem can be solved more efficiently. An encoding of a FPS scheduler using two clocks is presented in [11].

Scheduling is a well-established area. Various analysis methods have been published in the literature. For systems restricted to periodic tasks, algorithms such as rate monotonic scheduling are widely used and efficient methods for schedulability checking exist, see, e.g., [1]. These techniques can be used to handle non-periodic tasks. The standard way is to consider non-periodic tasks as periodic using the estimated *minimal* inter-arrival times as *task periods*. Clearly, the analysis based on such a task model would be pessimistic in many cases, e.g., a task set which is schedulable may be considered as non-schedulable as the inter-arrival times of the tasks may vary over time, that are not necessary minimal. Our work is more related to work on timed systems and scheduling.

An interesting work on relating classical scheduling theory to timed systems is the controller synthesis approach [12,13]. The idea is to achieve schedulability by construction. A general framework to characterize scheduling constraints as invariants and synthesize scheduled systems by decomposition of constraints is presented in [13]. However, algorithmic aspects are not discussed in these work. Timed automata has been used to solve non-preemptive scheduling problems mainly for job-shop scheduling [14–16]. These techniques specify predefined locations of an automaton as goals to achieve by scheduling and use reachability analysis to construct traces leading to the goal locations. The traces are used as schedules. There have been several work, e.g., [3,17,18] on using stopwatch automata to model preemptive scheduling problems. As the reachability analysis problem for stopwatch automata is undecidable in general [19], there is no guarantee for termination for the analysis without the assumption that task preemptions occur only at integer points. The idea of subtractions on timers with integers, was first proposed by McManis and Varaiya in [3]. In general, the class of timed automata with subtractions is undecidable, which is shown in [20]. In this paper, we have identified a decidable class of updatable automata, which is precisely what we need to solve scheduling problems without assuming that preemptions occur only at integer points.

Acknowledgement: The work is partially supported by EU through the CREDO project.

References

[1] G. C. Buttazzo, Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications, Kluwer Academic Publishers, 1997.

[2] C. Ericsson, A. Wall, W. Yi, Timed automata as task models for event-driven systems, in: Proc. of the 6*th* International Conference on Real-Time Computing Systems and Applications, IEEE Computer Society Press, 1999.

[3] J. McManis, P. Varaiya, Suspension automata: a decidable class of hybrid automata, in: Proc. of the 6*th* International Conference on Computer-Aided Verification, no. 818 in Lecture Notes in Computer Science, Springer–Verlag, 1994, pp. 105–117.

[4] K. G. Larsen, P. Pettersson, W. Yi, Compositional and symbolic model-checking of real-time systems, in: Proc. of 16*th* IEEE Real-Time Systems Symposium, IEEE Computer Society Press, 1995, pp. 76–89.

[5] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183–235.

[6] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools, in: W. Reisig, G. Rozenberg (Eds.), In Lecture Notes on Concurrency and Petri Nets, no. 3098 in Lecture Notes in Computer Science, Springer–Verlag, 2004.

[7] T. Henzinger, P. Kopke, A. Puri, P. Varaiya, What's decidable about hybrid automata?, Journal of Computer and System Sciences 57 (1998) 94–124.

[8] E. Fersman, P. Pettersson, W. Yi, Timed automata with asynchronous processes: Schedulability and decidability, in: J.-P. Katoen, P. Stevens (Eds.), Proc. of the 8*th* International Conference on Tools and Algorithms for the Construction and Analysis of Systems, no. 2280 in Lecture Notes in Computer Science, Springer–Verlag, 2002, pp. 67–82.

[9] P. Krčál, W. Yi, Decidable and undecidable problems in schedulability analysis using timed automata, in: K. Jensen, A. Podelski (Eds.), Proc. of TACAS'04, Barcelona, Spain., Vol. 2988 of Lecture Notes in Computer Science, Springer–Verlag, 2004, pp. 236–250.

[10] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, W. Yi, TIMES - a tool for modelling and implementation of embedded systems, in: Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, no. 2280 in Lecture Notes in Computer Science, Springer–Verlag, 2002.

[11] E. Fersman, L. Mokrushin, P. Pettersson, W. Yi, Schedulability analysis of fixed-priority systems using timed automata., Theor. Comput. Sci. 354 (2) (2006) 301–317.

[12] K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, S. Yovine, A framework for scheduler synthesis, in: Proc. of the 20*th* IEEE Real-Time Systems Symposium, IEEE Computer Society Press, 1999, pp. 154–163.

[13] K. Altisen, G. Gößler, J. Sifakis, A methodology for the construction of scheduled systems, in: Proc. of Formal Techniques in Real-Time and Fault Tolerant Systems, no. 1926 in Lecture Notes in Computer Science, Springer–Verlag, 2000, pp. 106–120.

[14] Y. Abdeddaim, O. Maler, Job-shop scheduling using timed automata, in: Proc. of 13*th* Conference on Computer Aided Verification, no. 2102 in Lecture Notes in Computer Science, Springer–Verlag, 2001.

[15] A. Fehnker, Scheduling a steel plant with timed automata, in: Proc. of the 6*th* International Conference on Real-Time Computing Systems and Applications, IEEE Computer Society Press, 1999.

[16] T. Hune, K. G. Larsen, P. Pettersson, Guided Synthesis of Control Programs using Uppaal, Nordic Journal of Computing 8 (1) (2001) 43–64.

[17] J. Corbett, Modeling and analysis of real-time ada tasking programs, in: Proc. of 15*th* IEEE Real-Time Systems Symposium, IEEE Computer Society Press, 1994, pp. 132–141.

[18] F. Cassez, F. Laroussinie, Model-checking for hybrid systems by quotienting and constraints solving, in: Proc. of the 12*th* International Conference on Computer-Aided Verification, no. 1855 in Lecture Notes in Computer Science, Springer–Verlag, 2000, pp. 373–388.

[19] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine, The algorithmic analysis of hybrid systems, Theoretical Computer Science 138 (1) (1995) 3–34.

[20] P. Bouyer, C. Dufourd, E. Fleury, A. Petit, Are timed automata updatable?, in: Proc. of the 12*th* International Conference on Computer-Aided Verification, Vol. 1855 of Lecture Notes in Computer Science, Springer–Verlag, 2000.

[21] K. G. Larsen, W. Yi, Time-abstracted bisimulation: Implicit specifications and decidability, Information and Computation 134 (2) (1997) 75–101.

A Appendix

Proof of Lemma 1. Assume $u \approx v$. To prove the first two clauses, we use the known fact on the standard region equivalence $\sim$, that $u{\sim}v$ implies that for all t, $u+t \sim v+t'$ for some real number t' such that $\lfloor t \rfloor = \lfloor t' \rfloor$ and $u[x \mapsto 0] \sim v[x \mapsto 0]$ for a clock x. Proofs can be found in the literature, e.g., [21]. Assume $g \in \mathcal{G}$ and g is in the form $x - y \bowtie N$. We have two cases:

(1) First, assume $u+t \models g$, that is, $u(x+t) - u(y+t) \bowtie N$. This implies $u(x) - u(y) \bowtie N$. Thus, $u \models g$. Because $u \approx v$, we also have $v \models g$. As $v \models g$ implies $v(x+t) - v(y+t) \bowtie N$ for any real t, we have $v + t' \models g$.

(2) Second, assume $u[x \mapsto 0] \models g$, that is, $-u(y) \bowtie n$. As $u{\sim}v$, $-v(y) \bowtie N$ by definition of $\sim$. That is, $v[x \mapsto 0] \models g$. The case for $u[y \mapsto 0] \models g$ is similar.

Therefore, we conclude the first two clauses in the lemma.

Now we prove the third clause. Assume that $u{\sim}v$, and for each clock x, $C \leq u(x) \leq C_x$. From $u{\sim}v$, we have $C \leq v(x) \leq C_x$. We have three cases to check:

(1) (The integer parts of clock values in $u(x-C)$ and $v(x-C)$). Because $u{\sim}v$, we have $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$. As $C \leq u(x) \leq C_x$, $\lfloor u(x-C)(x) \rfloor = \lfloor v(x-C)(x) \rfloor$. By definition, we have $\lfloor u(x-C)(y) \rfloor = \lfloor v(x-C)(y) \rfloor$ for all clocks y. This proves that the integer parts of clock values in $u(x-C)$ and $v(x-C)$ are equal.

(2) (The fractional parts of clock values in $u(x-C)$ and $v(x-C)$). As the subtraction operation on a clock only changes the integer part of the clock, we have, for all clocks y and z,
   (a) $\{u(x-C)(y)\} = 0$ iff $\{v(x-C)(y)\} = 0$ and
   (b) $\{u(x-C)(y)\} \leq \{u(x-C)(z)\}$ iff $\{v(x-C)(y)\} \leq \{v(x-C)(z)\}$

(3) (The diagonal constraints). Assume that g is in the form $x - y \bowtie N$ and $u(x-C) \models g$, i.e., $u(x) - C - u(y) \bowtie N$. We need to establish that $v(x) - C - v(y) \bowtie n$.

   Let $u(x) = \lfloor u(x) \rfloor + \{u(x)\}$, $u(y) = \lfloor u(y) \rfloor + \{u(y)\}$ and $M = \lfloor u(x) \rfloor - \lfloor u(y) \rfloor$. As $u{\sim}v$, $C \leq u(x) \leq C_x$ and $C \leq v(x) \leq C_x$, we have $\lfloor v(x) \rfloor - \lfloor v(y) \rfloor = M$. Now we need to prove that $\{u(x)\} - \{u(y)\} \bowtie N + C - M$ implies $\{v(x)\} - \{v(y)\} \bowtie N + C - M$.

   Consider the three cases of $\bowtie$:
   (a) $\{u(x)\} - \{u(y)\} = N + C - M$. It must be the case that $N + C - M = 0$. By $u{\sim}v$, $\{v(x)\} - \{v(y)\} = 0$.
   (b) $\{u(x)\} - \{u(y)\} < N + C - M$. As $N + C - M$ is an integer and $\{u(x)\} - \{u(y)\}$ is a real number within the open interval $(-1, 1)$, we must have $N + C - M \geq 0$. In case $N + C - M = 0$, we have $\{u(x)\} - \{u(y)\} < 0$ which implies $\{v(x)\} - \{v(y)\} < 0$ from $u{\sim}v$.

In case $N + C - M > 0$, we have immediately $\{v(x)\} - \{v(y)\} < N + C - M$.

(c) $\{u(x)\} - \{u(y)\} > N + C - M$. It is to prove $\{v(y)\} - \{v(x)\} < -N - C + M$ under the assumption $\{u(y)\} - \{u(x)\} < -N - C + M$. This is similar to the above case.

□

Example encoding of scheduling policies. For EDF, we need to compare the remaining relative deadlines. This information is kept in the clocks x^d_{ij} for all i, j such that $\mathsf{Status}(ij) \neq \mathsf{free}$. We assume that EDF handles the task instances with the same remaining deadline in the FIFO order. Let us denote the first task instance in the queue as p_{ef} and the second one as p_{gh}. Then

- $\mathsf{Head}(gh)$ is true if p_{ef} finishes its computation,
- $\mathsf{Head}(ef)$ is true if p_{mn} is released and $x^d_{ef} \geq D_e - D_m$,
- $\mathsf{Head}(mn)$ is true if p_{mn} is released and $x^d_{ef} < D_e - D_m$, and
- $\mathsf{Head}(mn)$ is false otherwise.

$\mathsf{Remove}(ef)$ simply removes the first task instance from the queue when p_{ef} finishes. When a new instance p_{mn} arrives, it is inserted by $\mathsf{Insert}(mn)$ at the head of the queue if $\mathsf{Head}(mn)$, at the tail of the queue if $x^d_{kl} \geq D_k - D_m$ for all k, l such that $\mathsf{Status}(kl) \neq free$, and between the neighbouring task instances p_{ij} and p_{kl} if $x^d_{kl} \geq D_k - D_m$ and $x^d_{kl} < D_i - D_m$.

Note that $\mathsf{Head}(kl)$ is true for exactly one pair k, l and that $\mathsf{Head}(kl)$, $\mathsf{Remove}(kl)$, and $\mathsf{Insert}(kl)$ correspond to the EDF scheduling policy if the value of the clocks x^d_{ij} is equal to $D_i - d_{ij}$ for each task instance p_{ij}.

For SJF, we need to compare the remaining (worst case) computation times. This information is kept in the clocks x^c_{ij}. We assume that SJF handles the task instances with the same remaining computation time in the FIFO order. Let us denote the first task instance in the queue as p_{ef} and the second one as p_{gh}. Then

- $\mathsf{Head}(gh)$ is true if p_{ef} finishes its computation,
- $\mathsf{Head}(ef)$ is true if p_{mn} is released and $x^c_{ef} \geq W_e - W_m$,
- $\mathsf{Head}(mn)$ is true if p_{mn} is released and $x^c_{ef} < W_e - W_m$, and
- $\mathsf{Head}(mn)$ is false otherwise.

$\mathsf{Remove}(ef)$ simply removes the first task instance from the queue when p_{ef} finishes. When a new instance p_{mn} arrives, it is inserted by $\mathsf{Insert}(mn)$ at the head of the queue if $\mathsf{Head}(mn)$, at the tail of the queue if $W_m \geq W_k$ for all k, l such that $\mathsf{Status}(kl) = \mathsf{released}$ and $W_m - W_k \geq x^c_{ef} - x^c_{kl}$ for all k, l such that

$\mathsf{Status}(kl) = \mathsf{preempted}$. It is inserted between the neighbouring task instances p_{ij} and p_{kl} if

- $W_m \geq W_i$ and $W_m < W_k$, where $\mathsf{Status}(ij) = \mathsf{Status}(kl) = \mathsf{released}$,
- $W_m - W_i \geq x^c_{ef} - x^c_{ij}$ and $W_m < W_k$, where $\mathsf{Status}(ij) = \mathsf{preempted}$, $\mathsf{Status}(kl) = \mathsf{released}$, and $\mathsf{Preempted\_by}(ij)(ef)$,
- $W_m - W_i \geq x^c_{ef} - x^c_{ij}$ and $W_m - W_k < x^c_{ij} - x^c_{kl}$, where $\mathsf{Status}(ij) = \mathsf{Status}(kl) = \mathsf{preempted}$ and $\mathsf{Preempted\_by}(ij)(ef)$,
- $x^c_{ij} \geq W_e - W_m$ and $W_m < W_k$, where $\mathsf{Status}(ij) = \mathsf{running}$ and $\mathsf{Status}(kl) = \mathsf{released}$, or
- $x^c_{ij} \geq W_e - W_m$ and $W_m - W_k < x^c_{ij} - x^c_{kl}$, where $\mathsf{Status}(ij) = \mathsf{running}$ and $\mathsf{Status}(kl) = \mathsf{preempted}$.

Note that $\mathsf{Head}(kl)$ is true for exactly one pair k, l and that $\mathsf{Head}(kl)$, $\mathsf{Remove}(kl)$, and $\mathsf{Insert}(kl)$ correspond to SJF if $w_{ij} = W_i - v(x^c_{ij})$ for each running task instance p_{ij} and $w_{ij} = W_i - (v(x^c_{ij}) - v(x^c_{kl}))$ for all preempted task instances p_{ij} where $\mathsf{Preempted\_by}(ij)(kl)$].