Algorithmica manuscript No.
(will be inserted by the editor)

Feasibility Analysis of Sporadic Real-Time Multiprocessor Task Systems

Vincenzo Bonifaci · Alberto Marchetti-Spaccamela

the date of receipt and acceptance should be inserted later

Abstract We give the first algorithm for testing the feasibility of a system of sporadic real-time tasks on a set of identical processors, solving an open problem in the area of multiprocessor real-time scheduling [S. Baruah and K. Pruhs, Journal of Scheduling, 2009]. We also investigate the related notion of schedulability and a notion that we call online feasibility. Finally, we show that discrete-time schedules are as powerful as continuous-time schedules, which answers another open question in the above mentioned survey.

Keywords Sporadic task system · Multiprocessor · Real-time scheduling · Feasibility test · Schedulability test

1 Introduction

As embedded microprocessors become more and more common, so does the need to design systems that are guaranteed to meet deadlines in applications that are safety critical, where missing a deadline might have severe consequences. In such a real-time system, several tasks may need to be executed on a multiprocessor platform and a scheduling policy needs to decide which tasks should be active in which intervals, so as to guarantee that all deadlines are met.

The *sporadic task model* is a model of recurrent processes in hard real-time systems that has received great attention in the last years; see for example

A preliminary version of this work appeared in *Proceedings of the 18th European Symposium on Algorithms*, Lecture Notes in Computer Science, Springer, Berlin, 2010, pp. 230–241.

V. Bonifaci
Max-Planck Institut für Informatik, Saarbrücken, Germany
E-mail: bonifaci@mpi-inf.mpg.de

A. Marchetti-Spaccamela
Sapienza Università di Roma, Rome, Italy
E-mail: alberto@dis.uniroma1.it

[2,7] and references therein. A sporadic task $\tau_i = (C_i, D_i, P_i)$ is characterized by a worst-case execution time C_i, a relative deadline D_i, and a minimum interarrival separation P_i. Such a sporadic task generates a potentially infinite sequence of jobs: each job arrives at an unpredictable time, after the minimum separation P_i from the last job of the same task has elapsed; it has an execution requirement less than or equal to C_i and a deadline that occurs D_i time units after its arrival time. A sporadic task system $\mathcal{T}$ is a collection of such sporadic tasks. Since the actual interarrival times can vary, there are infinitely many job sequences that can be generated by $\mathcal{T}$.

We are interested in designing algorithms that tell us when a given sporadic task system can be feasibly scheduled on a set of m identical processors, where we allow any job to be interrupted and resumed later on another processor at no penalty. The problem can be formulated in several ways:

- *Feasibility*: is it possible to feasibly schedule on m processors any job sequence that can be generated by $\mathcal{T}$?
- *Online feasibility*: is there an online algorithm that can feasibly schedule on m processors any job sequence that can be generated by $\mathcal{T}$?
- *Schedulability*: does the given online algorithm Alg feasibly schedule on m processors any job sequence that can be generated by $\mathcal{T}$?

The questions are formalized in Section 2. Here we only remark that in general they may have different answers.

Previous work. Most of the previous work in the context of sporadic real-time feasibility testing has focused on the case of a single processor [6]. The seminal paper by Liu and Layland [16] gave a best possible fixed-priority algorithm for the case where deadlines equal periods (a fixed-priority algorithm assigns a distinct priority to each task and then – at each time instant – schedules the available job from the task with highest priority). It is also known that the Earliest Deadline First (EDF) algorithm, that schedules at any time the job with the earliest absolute deadline, is optimal in the sense that for any sequence of jobs it produces a valid schedule whenever a valid schedule exists [10]. Because EDF is an online algorithm, this implies that the three questions of feasibility, of online feasibility and of schedulability with respect to EDF are equivalent in the single processor case. It was known for some time that EDF-schedulability could be tested in exponential time and more precisely that the problem is in coNP [8]. The above results triggered a significant research effort within the scheduling community and many results have been proposed for specific algorithms and/or special cases; nonetheless, we remark that the precise complexity of the feasibility problem for a single processor remained open for a long time and that only recently it has been proved coNP-complete [12].

The case of multiple processors is far from being as well understood as the single processor case. For starters, EDF is no longer optimal – it is not hard to construct feasible task systems for which EDF fails, as soon as $m \geq 2$. Another important difference with the single processor case is that here clairvoyance

does help the scheduling algorithm: there exists a task system that is feasible, but for which no online algorithm can produce a feasible schedule on every job sequence [13]. Thus, the notions of feasibility and online feasibility are distinct.

On the positive side, some results are known for special cases of the feasibility problem; however we remark that *no test – of whatsoever complexity – is known* that correctly decides the feasibility or the online feasibility of a task system. This holds even for *constrained-deadline* systems, in which deadlines do not exceed periods. The question of designing such a test has been listed as one of the main algorithmic problems in real-time scheduling [7].

Regarding schedulability, many schedulability tests are known for specific algorithms (see [2] and references therein), but, to the best of our knowledge, the only general test available is a test that requires exponential space [3].

Our results. We study the three above problems in the context of sporadic multiprocessor systems, and we provide new results for each of them.

For the feasibility problem, we give the first correct test, thus answering [7, Open Problem 3]. The test has high complexity, but it has the interesting consequence that a job sequence that witnesses the infeasibility of a task system $\mathcal{T}$ has without loss of generality length at most doubly exponential in the bitsize of $\mathcal{T}$.

We then give the first correct test for the online feasibility problem. The test has exponential time complexity and is constructive: if a system is deemed online feasible, then an optimal online algorithm for it can be constructed (in the same time bound). Moreover, this optimal algorithm is without loss of generality *memoryless*: its decisions depend only on the current (finite) state and not on the entire history up to the decision point (see Section 2 for a formal definition).

For the schedulability problem, we provide a general schedulability test showing that the schedulability of a system by any memoryless algorithm can be tested in polynomial space. This improves the result of Baker and Cirinei [3], that provided an exponential space test for essentially the same class of algorithms.

All the above results, that are derived for constrained-deadline systems where $D_i \leq P_i$ for all i, can be extended to the arbitrary-deadline case in which deadlines may exceed periods, at the expense of increasing some of the complexity bounds. The extension is discussed in Section 4.

We finally consider the issue of discrete time schedules versus continuous time schedules. The above results are derived with the assumption that the time line is divided into indivisible time slots and preemptions can occur only at integral points, that is, the schedule has to be discrete. In a continuous schedule, time is not divided into discrete quanta and preemptions may occur at any time instant. We show that in a sporadic task system a discrete schedule exists whenever a continuous schedule does, thus showing that the discrete time assumption is without loss of generality. Such an equivalence was known for the single processor setting [8]; however, the proof relied on the optimality of the EDF algorithm and thus did not extend to multiprocessor systems. In

fact, this problem was also cited among the relevant open problems in real-time scheduling [7, Open Problem 5].

Our main conceptual contribution is to show how the feasibility problem, the online feasibility problem and the schedulability problem can be cast as the problem of deciding the winner in certain games of infinite duration played on a finite graph. We then use tools from the theory of games to decide who has a winning strategy. In particular, in the case of the feasibility problem we have a game of imperfect information where one of the players does not see the moves of the opponent, a so-called *blindfold game* [18]. This can be reformulated as a one-player (i.e., solitaire) game on an exponentially larger graph and then solved via a reachability algorithm. However, a technical complication is that in our model a job sequence and a schedule can both have infinite length, which when the system is feasible makes the construction of a feasible schedule challenging. We solve this complication by an application of König's Infinity Lemma from graph theory [11, p. 200]. This is the technical ingredient that, roughly speaking, allows us to reduce the job sequences from infinite length to finite length and ultimately to obtain the equivalence between continuous and discrete schedules.

The power of our new approach is its generality: it can be applied to all three problems, and at the same time it yields proofs that are not technically too complicated. We hope that this approach might be useful to answer similar questions for other real-time scheduling problems.

Organization. The remainder of the paper is structured as follows. In Section 2 we formally define the model and set up some useful notation. In Section 3 we describe and analyze our algorithms for feasibility and schedulability analysis. Section 4 discusses the extension of the results to the case where deadlines may exceed periods. The equivalence between continuous and discrete schedules is treated in Section 5, and we finish with some concluding remarks in Section 6.

2 Definitions

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ and $[n] = \{1, 2, \ldots, n\}$. Given a set X, with $\binom{X}{k}$ we denote the set of all k-subsets of X.

Consider a task system $\mathcal{T}$ with n tasks, and m processors; without loss of generality, $m \leq n$. Each task i is described by three parameters: a worst-case *execution time* C_i, a *relative deadline* D_i, and a *minimum interarrival time* P_i. We assume these parameters to be positive integers and that $D_i \leq P_i$ for all i (the latter assumption is dropped in Section 4). Without loss of generality, $C_i \leq D_i$ for all i, otherwise the system is clearly infeasible.

Let $\mathbf{C} := \times_{i=1}^{n}([C_i] \cup \{0\})$, $\mathbf{D} := \times_{i=1}^{n}([D_i] \cup \{0\})$, $\mathbf{P} := \times_{i=1}^{n}([P_i] \cup \{0\})$, $\mathbf{0} := (0)_{i=1}^{n}$. A *job sequence* is a function $\sigma : \mathbb{N} \to \mathbf{C}$. The interpretation is that $\sigma(t) = (\sigma_i(t))_{i=1}^{n}$ iff, for each i with $\sigma_i(t) > 0$, a new job from task i is released at time t with execution time $\sigma_i(t)$, and no new job from task i is

released if $\sigma_i(t) = 0$. A *legal* job sequence has the additional property that for any distinct $t, t' \in \mathbb{N}$ and any i, if $\sigma_i(t) > 0$ and $\sigma_i(t') > 0$, then $|t - t'| \geq P_i$. A job sequence is *finite* if $\sigma(t') = \mathbf{0}$ for all t' greater or equal to some $t \in \mathbb{N}$; in this case, we say that the sequence has *length* t.

Let $\mathbf{S} := \cup_{k=0}^{m} \binom{[n]}{k}$. A *schedule* is a function $S : \mathbb{N} \to \mathbf{S}$; we interpret $S(t)$ as the set of those k tasks ($0 \leq k \leq m$) that are being processed from time t to time $t + 1$ [1]. We allow that $S(t)$ contains a task i even when there is no pending job from i at time t; in that case there is no effect (this is formalized below).

A *backlog configuration* is an element of $\mathbf{B} := \mathbf{C} \times \mathbf{D} \times \mathbf{P}$. At time t, a backlog configuration[2] $(c_i, d_i, p_i)_{i=1}^{n} \in \mathbf{B}$ will denote the following:

- $c_i \in [C_i] \cup \{0\}$ is the *remaining execution time* of the unique pending job from task i, if any; if there is no pending job from task i, then $c_i = 0$;
- $d_i \in [D_i] \cup \{0\}$ is the *remaining time to deadline* of the unique pending job from task i, if any; if there is no pending job from task i, or the deadline has already passed, then $d_i = 0$;
- $p_i \in [P_i] \cup \{0\}$ is the *minimum remaining time to the next activation* of task i, that is, the minimum p_i such that a new job from task i could be legally released at time $t + p_i$.

A configuration $(c_i, d_i, p_i)_{i=1}^{n} \in \mathbf{B}$ is a *failure configuration* if for some task i, $c_i > 0$ and $d_i = 0$.

Remark 1 The set $\mathbf{B}$ is finite, and its size is $2^{O(s)}$, where s is the input size of $\mathcal{T}$ (number of bits in its binary encoding).

Given a legal job sequence σ and a schedule S, we define in the natural way an infinite sequence of backlog configurations $\langle \sigma, S \rangle := b_0 b_1 \ldots$. The initial configuration is $b_0 := (0, 0, 0)_{i=1}^{n}$, and given a backlog configuration $b_t = (c_i, d_i, p_i)_{i=1}^{n}$, its successor configuration $b_{t+1} = (c'_i, d'_i, p'_i)_{i=1}^{n}$ is obtained as follows:

- if $\sigma_i(t) > 0$, then $c'_i = \sigma_i(t) - x_i$, where x_i is 1 if $i \in S(t)$, and 0 otherwise; moreover, $d'_i = D_i - 1$ and $p'_i = P_i - 1$;
- if $\sigma_i(t) = 0$, then $c'_i = \max(c_i - x_i, 0)$, where x_i is defined as above; moreover, $d'_i = \max(d_i - 1, 0)$ and $p'_i = \max(p_i - 1, 0)$.

We can now define a schedule S to be *feasible for* σ if no failure configuration appears in $\langle \sigma, S \rangle$. Finally, a task system $\mathcal{T}$ is *feasible* when every legal job sequence admits a feasible schedule. Stated otherwise, a task system is not feasible when there is a legal job sequence for which no schedule is feasible. We call such a job sequence a *witness* of infeasibility.

A *deterministic online algorithm* Alg is a sequence of functions:

$$\mathrm{Alg}_t : \mathbf{C}^{t+1} \to \mathbf{S}, \qquad t = 0, 1, 2, \ldots$$

[1] Since $D_i \leq P_i$, there can be at most one pending job from task i. The case where D_i can be larger than P_i is considered in Section 4.

[2] For notational convenience, here we have reordered the variables so as to have n-tuples of triples, instead of triples of n-tuples.

By applying an algorithm Alg to a job sequence σ, one obtains the schedule S defined by $S(t) = \mathrm{Alg}_t(\sigma(0), \ldots, \sigma(t))$. Then Alg feasibly schedules σ whenever S does. A *memoryless* algorithm is a single function $\mathrm{Malg} : \mathbf{B} \times \mathbf{C} \to \mathbf{S}$; it is a special case of an online algorithm in which the scheduling decisions at time t are based only on the current backlog configuration and on the tasks that have been activated at time t.

Finally, a task system $\mathcal{T}$ is *online feasible* if there is a deterministic online algorithm Alg such that every legal job sequence from $\mathcal{T}$ is feasibly scheduled by Alg. We then say that Alg is *optimal* for $\mathcal{T}$, and that $\mathcal{T}$ is *schedulable* by Alg. Online feasibility implies feasibility, but the converse fails: there is a task system that is feasible, but that does not admit any optimal online algorithm [13].

König's Infinity Lemma. A *ray* is an infinite graph (V, E) of the form

$$V = \{x_0, x_1, x_2, ...\},\ E = \{(x_0, x_1), (x_1, x_2), (x_2, x_3), \ldots\}.$$

Lemma 1 *Let $Q^0, Q^1, \ldots$ be an infinite sequence of disjoint nonempty finite sets of nodes, and let G be a graph on their union. Assume that every node q in a set Q^t with $n \geq 1$ has a predecessor q' in Q^{t-1}, so that (q', q) is an arc of G. Then G contains a ray $q_0, q_1, \ldots$ with $q_t \in Q^t$ for all t.*

Proof See for example [11, Lemma 8.1.2] (the result is stated there in terms of undirected graphs, but the proof works equally well for the directed case.) □

3 Algorithms for feasibility and schedulability analysis

3.1 Feasibility

We first model the process of scheduling a task system as a game between two players over infinitely many rounds. At round $t = 0, 1, 2, \ldots$, the first player (the "adversary") selects a certain set of tasks to be activated. Then the second player (acting as the scheduler) selects a set of tasks to be processed, and so on. The game is won by the first player if a failure configuration is eventually reached.

In order to capture the definition of feasibility correctly, the game must proceed so that the adversary has no information at all on the moves of the scheduler; in other words, the job sequence must be constructed obliviously from the schedule. This is because *if the task system is infeasible, then a single witness job sequence must fail all possible schedules simultaneously.* Models of such games, where the first player has no information on the moves of the opponent, have been studied in the literature under the name of *blindfold games* [18]. One approach to solving these games is to construct a larger one-player game, in which each state encodes all positions that are compatible with at least one sequence of moves for the second player.

We now proceed to give the details of the construction. The guiding intuition is that the scheduling process can be summarized by looking only at a finite set of counters, including the "time to deadline" counters, the "time to next earliest arrival" counters, as well as the internal state of the scheduler (an element of the set $\mathbf{C}$), which specifies the remaining execution times of the jobs. The actual internal scheduler state is unknown to the adversary, but since it is finite, it will be possible to consider it implicitly by enumeration.

Given a task system $\mathcal{T}$, we build a bipartite graph $G^+(\mathcal{T}) = (V_1, V_2, A)$. Nodes in V_1 (V_2) will correspond to decision points for the adversary (scheduler). A node in V_1 or V_2 will encode mainly two kinds of information: (1) the counters that determine time to deadlines and next earliest arrival dates; and (2) the set of all plausible remaining execution times of the scheduler.

Let $\mathbf{B}^+ := \mathbf{D} \times \mathbf{P} \times 2^{\mathbf{C}}$. Each of V_1 and V_2 is a copy of $\mathbf{B}^+$, so each node of V_1 is identified by a distinct element from $\mathbf{B}^+$, and similarly for V_2. We now specify the arcs of $G^+(\mathcal{T})$. Consider an arbitrary node $v_1 \in V_1$ and let $((d_i, p_i)_{i=1}^n, Q)$ be its identifier, where $Q \in 2^{\mathbf{C}}$. Its successors in $G^+(\mathcal{T})$ are all nodes $v_2 = ((d'_i, p'_i)_{i=1}^n, Q') \in V_2$ for which there is a tuple $(k_i)_{i=1}^n \in \mathbf{C}$ such that:

1. $p_i = 0$ for all $i \in \mathrm{supp}(k)$, where $\mathrm{supp}(k) = \{i : k_i > 0\}$ (this ensures that each task in k can be activated);
2. $p'_i = P_i$, and $d'_i = D_i$ for all $i \in \mathrm{supp}(k)$ (activated jobs cannot be reactivated before P_i time units);
3. $p'_i = p_i$ and $d'_i = d_i$ for all $i \notin \mathrm{supp}(k)$ (counters of other tasks are not affected);
4. each $(c'_i)_{i=1}^n \in Q'$ is obtained from some $(c_i)_{i=1}^n \in Q$ in the following way: $c'_i = k_i$ for all $i \in \mathrm{supp}(k)$, and $c'_i = c_i$ for all $i \notin \mathrm{supp}(k)$ (in every possible scheduler state, the remaining execution time of each activated job is set to the one prescribed by k);
5. Q' contains all $(c'_i)_{i=1}^n$ that satisfy Condition 4.

Now consider an arbitrary node $v_2 \in V_2$, say $v_2 = ((d_i, p_i)_{i=1}^n, Q)$. The only successor of v_2 will be the unique node $v_1 = ((d'_i, p'_i)_{i=1}^n, Q') \in V_1$ such that:

1. $d'_i = \max(d_i - 1, 0)$, $p'_i = \max(p_i - 1, 0)$ for all $i \in [n]$ (this models a "clock-tick");
2. for each $(c'_i)_{i=1}^n \in Q'$, there are an element $(c_i)_{i=1}^n \in Q$ and some $S \in \mathbf{S}$ such that $c'_i = \max(c_i - 1, 0)$ for all $i \in S$ and $c'_i = c_i$ for all $i \notin S$ (each new possible state of the scheduler is obtained from some old state after the processing of at most m tasks);
3. for each $(c'_i)_{i=1}^n \in Q'$, one has, for all i, $c'_i = 0$ whenever $d'_i = 0$ (this ensures that the resulting scheduler state is valid);
4. Q' contains all $(c'_i)_{i=1}^n$ that satisfy Condition 2 and Condition 3.

That is, the only successor to v_2 is obtained by applying all possible decisions by the scheduler and then taking Q' to be the set of all possible (valid) resulting scheduler states. Notice that because we only keep the valid states (Condition 3), the set Q' might be empty. In this case we say that the node v_1 is a *failure*

state; it corresponds to some deadline having been violated. Also notice that any legal job sequence σ induces an alternating walk in the bipartite graph $G^+(\mathcal{T})$ whose $(2t+1)$-th arc corresponds to $\sigma(t)$.

Finally, the *initial state* is the node $v_0 \in V_1$ for which $d_i = p_i = 0$ for all i, and for which the only possible scheduler state is $\mathbf{0}$. (See Figure 1 on page 9 for a partial illustration of the construction in the case of the task system $\mathcal{T} = ((1,2,2),(2,2,2))$, for $m = 1$.) Note that, given two nodes of $G^+(\mathcal{T})$, it is easy to check their adjacency, in time polynomial in $|\mathbf{B}^+|$.

Definition 1 For a legal job sequence σ, the set of *possible valid scheduler states* at time t is the set of all $(c_i)_{i=1}^n \in \mathbf{C}$ for which there exists a schedule S such that (i) $\langle \sigma, S \rangle = b_0 b_1 b_2 \ldots$ with no configuration $b_0, b_1, \ldots, b_t$ being a failure configuration, and (ii) the first component of b_t is $(c_i)_{i=1}^n$. We denote this set by $\mathrm{valid}(\sigma, t)$.

The proofs of the following two lemmas are given as sketches in the sense that we omit a complete formal analysis; this is done only in order to avoid unnecessarily tedious statements, which in turn can be recovered fairly easily by referring back to the definitions.

Lemma 2 *Let $t \geq 0$ and let $((d_i, p_i)_{i=1}^n, Q) \in V_1$ be the node reached by following for $2t$ steps the walk induced by σ in the graph $G^+(\mathcal{T})$. Then $Q = \mathrm{valid}(\sigma, t)$.*

Proof (sketch) By induction on t. When $t = 0$ the claim is true because the only possible scheduler state is the $\mathbf{0}$ state. For larger t it follows from how we defined the successor relation in $G^+(\mathcal{T})$ (see in particular the definition of Q'). □

Lemma 3 *Task system $\mathcal{T}$ is infeasible if and only if, in the graph $G^+(\mathcal{T})$, some failure state is reachable from the initial state.*

Proof (sketch) If there is a path from the initial state to some failure state, by Lemma 2 we obtain a legal job sequence σ that witnesses that for some t, $\mathrm{valid}(\sigma, t) = \emptyset$, that is, there is no valid scheduler state for σ at time t; so there cannot be any feasible schedule for σ.

Conversely, if no failure state is reachable from the initial state, for any legal job sequence σ one has $\mathrm{valid}(\sigma, t) \neq \emptyset$ for all t by Lemma 2. This immediately implies that no *finite* job sequence can be a witness of infeasibility. We also need to exclude witnesses of infinite length. To do this, we apply König's Infinity Lemma (Lemma 1). Consider the infinite walk induced by σ in $G^+(\mathcal{T})$ and the corresponding infinite sequence of nonempty sets of possible valid scheduler states $Q^0, Q^1, \ldots$, where $Q^t := \mathrm{valid}(\sigma, t)$. Each scheduler state $q \in Q^t$ $(t \geq 1)$ has been derived by some scheduler state in $q' \in Q^{t-1}$ and so q and q' can be thought of as neighbors in an infinite graph on the disjoint union of $Q^0, Q^1, \ldots$ (see Figure 2). Then König's Lemma implies that there is a sequence $q_0 q_1 \ldots$ (with $q_t \in Q^t$) such that for all $t \geq 1$, q_t is a neighbor of q_{t-1}. This sequence defines a feasible schedule for σ. □

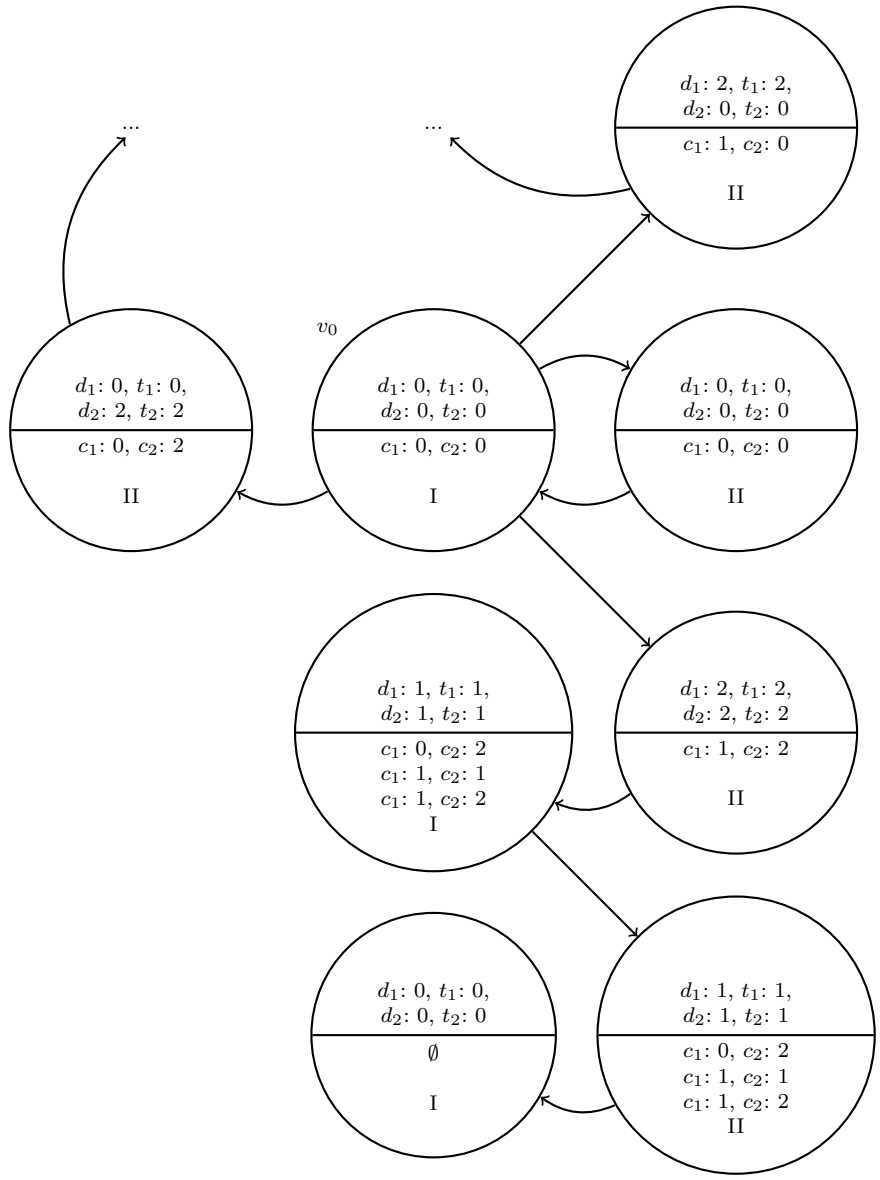

Fig. 1 A subgraph of the graph $G^+(\mathcal{T})$ for the task system $\mathcal{T} = ((1,2,2),(2,2,2))$ and $m = 1$. Nodes labeled with "I" are in V_1, nodes labeled with "II" are in V_2.

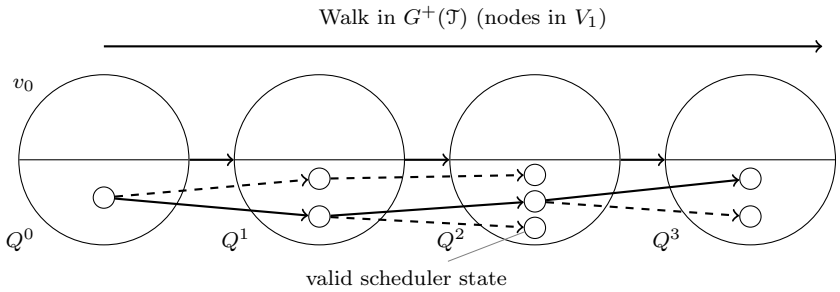

Fig. 2 Illustration of how König's Lemma applies to the proof of Lemma 3. A prefix of an infinite ray is shown in solid lines.

Algorithm 1 Algorithm for the feasibility problem

```
for all failure states v_f ∈ V_1 do
  if REACH(v_0, v_f, 2|B^+|) then
    return infeasible
  end if
end for
return feasible
```

Algorithm 2 REACH(x, y, k)

```
if k = 0 then
  return true if x = y, false if x ≠ y
end if
if k = 1 then
  return true if (x, y) ∈ A, false otherwise
end if
for all z ∈ V_1 ∪ V_2 do
  if REACH(x, z, ⌊k/2⌋) and REACH(z, y, ⌈k/2⌉) then
    return true
  end if
end for
return false
```

Theorem 1 *The feasibility problem for a sporadic constrained-deadline task system $\mathcal{T}$ can be solved in time $2^{2^{O(s)}}$, where s is the input size of $\mathcal{T}$. Moreover, if $\mathcal{T}$ is infeasible, there is a witness job sequence of length at most $2^{2^{O(s)}}$.*

Proof The graph has $2|\mathbf{B}^+| = 2^{2^{O(s)}}$ nodes, so the first part follows from Lemma 3 and the existence of linear-time algorithms for the reachability problem. The second part follows similarly from the fact that the witness sequence σ can be defined by taking $\sigma(t)$ as the set of task activations corresponding to the $(2t+1)$-th arc on the path from the initial state to the reachable failure state. □

We can in fact improve exponentially the amount of memory needed for the computation. The idea is to compute the state graph as needed, instead

of storing it explicitly (Algorithm 1). We enumerate all failure nodes; for each failure node v_f, we check whether there exists a path from v_0 to v_f in $G^+(\mathcal{T})$ by calling the subroutine REACH (Algorithm 2). This subroutine checks recursively whether there is a path from x to y of length at most k by trying all possible midpoints z. Some readers might recognize that REACH is nothing but Savitch's reachability algorithm [19]. This yields the following improvement.

Theorem 2 *The feasibility problem for a sporadic constrained-deadline task system $\mathcal{T}$ can be solved in space $2^{O(s)}$, where s is the input size of $\mathcal{T}$.*

Proof Any activation of Algorithm 2 needs $O(\log |\mathbf{B}^+|) = 2^{O(s)}$ space, and the depth of the recursion is at most $O(\log |\mathbf{B}^+|) = 2^{O(s)}$. □

3.2 Online feasibility

An issue with the notion of feasibility as studied in the previous section is that, when the task system turns out to be feasible, one is still left clueless as to how the system should be scheduled. The definition of online feasibility (see Section 2) addresses this issue. One might doubt whether the notion of online feasibility is really different from the notion of feasibility, but in fact recent work has shown that in a multiprocessor sporadic task system the two notions are not equivalent [13]. In other words, there is a particular task system $\mathcal{T}^*$ that is feasible, but that does not admit any optimal online algorithm; this also explains why the use of a nonconstructive result like König's Lemma was unavoidable in the previous section. In any event, it could be argued from a system design point of view that one should focus on the notion of online feasibility, rather than on the notion of feasibility. In this section we discuss an algorithm for testing online feasibility.

The idea is again to interpret the process as a game between the environment and the scheduler, with the difference that now the adversary can observe the current state of the scheduler (the remaining execution times). In other words, the game is no longer a blindfold game but a perfect-information game. We construct a graph $G(\mathcal{T}) = (V_1, V_2, A)$ where $V_1 = \mathbf{B}$ and $V_2 = \mathbf{B} \times \mathbf{C}$. The nodes in V_1 are decision points for the adversary (with different outgoing arcs corresponding to different tasks being activated) and the nodes in V_2 are decision points for the scheduler (different outgoing arcs corresponding to different sets of tasks being scheduled). There is an arc $(v_1, v_2) \in A$ if $v_2 = (v_1, k)$ for some tuple $k = (k_i)_{i=1}^n \in \mathbf{C}$ of jobs that can legally be released when the backlog configuration is v_1; notice the crucial fact that whether some tuple k can legally be released can be decided on the basis of the backlog configuration v_1 alone. There is an arc (v_2, v_1') if $v_2 = (v_1, k)$ and v_1' is a backlog configuration that can be obtained from v_1 after scheduling some subset of tasks; again this depends only on v_1 and k.

The details of the adjacency relation are as follows. Consider an arbitrary node $v_1 \in V_1$ with $v_1 = (c_i, d_i, p_i)_{i=1}^n$. Its successors in $G(\mathcal{T})$ are all nodes $v_2 = ((c_i', d_i', p_i')_{i=1}^n, (k_i)_{i=1}^n) \in V_2$ with $(k_i)_{i=1}^n \in \mathbf{C}$ such that:

1. $p_i = 0$ for all $i \in \mathrm{supp}(k)$, where $\mathrm{supp}(k) = \{i : k_i > 0\}$ (this ensures that each task in k can be activated);
2. $p'_i = P_i$, and $d'_i = D_i$ for all $i \in \mathrm{supp}(k)$ (activated jobs cannot be reactivated before P_i time units);
3. $p'_i = p_i$ and $d'_i = d_i$ for all $i \notin \mathrm{supp}(k)$ (counters of other tasks are not affected);
4. $c'_i = k_i$ for all $i \in \mathrm{supp}(k)$, and $c'_i = c_i$ for all $i \notin \mathrm{supp}(k)$ (the remaining execution time of each activated job is set to the one prescribed by k);

Now consider an arbitrary node $v_2 \in V_2$, say $v_2 = ((c_i, d_i, p_i)_{i=1}^n, (k_i)_{i=1}^n)$. Its successors in $G(\mathcal{T})$ are all nodes $v_1 = ((c'_i, d'_i, p'_i)_{i=1}^n) \in V_1$ for which there is some $S \in \mathbf{S}$ such that:

1. $d'_i = \max(d_i - 1, 0)$, $p'_i = \max(p_i - 1, 0)$ for all $i \in [n]$ (this models a "clock-tick");
2. $c'_i = \max(c_i - 1, 0)$ for all $i \in S$ and $c'_i = c_i$ for all $i \notin S$ (the new remaining execution times are obtained from the old remaining execution times by processing at most m tasks).

The game is now played with the adversary starting first in state $b_0 = (0, 0, 0)_{i=1}^n$. The two players take turns alternately and move from state to state by picking an outgoing arc from each state. The adversary wins if it can reach a state in V_1 corresponding to a failure configuration. The scheduler wins if it can prolong play indefinitely while never incurring in a failure configuration.

Lemma 4 *The first player has a winning strategy in the above game on $G(\mathcal{T})$ if and only if $\mathcal{T}$ is not online feasible. Moreover, if $\mathcal{T}$ is online feasible, then it admits an optimal memoryless deterministic online algorithm.*

Proof (sketch) If the first player has a winning strategy s, then for any online algorithm Alg, the walk in $G(\mathcal{T})$ obtained when player 1 plays according to s and player 2 plays according to Alg, ends up in a failure configuration. But then the job sequence corresponding to this walk in the graph (given by the odd-numbered arcs in the walk) defines a legal job sequence that is not feasibly scheduled by Alg.

If, on the other hand, the first player does not have a winning strategy, from the theory of two-player perfect-information games it is known (see for example [14,17]) that the second player has a winning strategy and that this can be assumed to be, without loss of generality, a deterministic strategy that depends only on the current state in V_2 (a so-called memoryless, or positional, strategy). Hence, for each node in V_2 it is possible to remove all but one outgoing arc so that in the remaining graph no failure configuration is reachable from b_0. The set of remaining arcs that leave V_2 implicitly defines a function from $V_2 = \mathbf{B} \times \mathbf{C}$ to $\mathbf{S}$, that is, a memoryless online algorithm, which feasibly schedules every legal job sequence of $\mathcal{T}$. □

Theorem 3 *The online feasibility problem for a sporadic constrained-deadline task system $\mathcal{T}$ can be solved in time $2^{O(s)}$, where s is the input size of $\mathcal{T}$. If $\mathcal{T}$ is online feasible, an optimal memoryless deterministic online algorithm for $\mathcal{T}$ can be constructed within the same time bound.*

Proof We first construct $G(\mathcal{T})$ in time polynomial in $|\mathbf{B} \times (\mathbf{B} \times \mathbf{C})| = 2^{O(s)}$. We then apply the following inductive algorithm to compute the set of nodes $W \subseteq V_1 \cup V_2$ from which player 1 can force a win; its correctness has been proved before (see for example [14, Proposition 2.18]). Define the set W_i as the set of nodes from which player 1 can force a win in at most i moves, so $W = \cup_{i \geq 0} W_i$. The set W_0 is simply the set of all failure configurations. The set W_{i+1} is computed from W_i as follows:

$$W_{i+1} = W_i \cup \{v_1 \in V_1 : (v_1, w) \in A \text{ for some } w \in W_i\} \\ \cup \{v_2 \in V_2 : w \in W_i \text{ for all } (v_2, w) \in A\}.$$

At any iteration either $W_{i+1} = W_i$ (and then $W = W_i$) or $W_{i+1} \setminus W_i$ contains at least one node. Since there are $2^{O(s)}$ nodes, this means that $W = W_k$ for some $k = 2^{O(s)}$. Because every iteration can be carried out in time $2^{O(s)}$, it follows that the set W can be computed within time $(2^{O(s)})^2 = 2^{O(s)}$. By Lemma 4, $\mathcal{T}$ is online feasible if and only if $b_0 \notin W$.

The second part of the claim follows from the second part of Lemma 4 and from the fact that a memoryless winning strategy for player 2 (that is, an optimal memoryless scheduler) can be obtained by selecting, for each node $v_2 \in V_2 \setminus W$, any outgoing arc that does not have an endpoint in W. □

3.3 Schedulability

In this section we show that the problem of deciding whether a sporadic task system is schedulable by some prescribed scheduling algorithm can be solved in polynomial space for all memoryless algorithms. This includes all commonly studied algorithms such as Earliest Deadline First, Least Laxity First, Fixed Priority, etc.; a non-memoryless algorithm would take into consideration the entire history and so its running time would degrade with the length of the job sequence, making it completely impractical. As we have seen in Lemma 4, an optimal memoryless algorithm always exists whenever the system is online feasible.

In the case of the schedulability problem, we observe that the construction of Section 3.1 can be applied in a simplified form, because for every node of the graph there is now *at most one* possible valid scheduler state, which can be determined by querying the scheduling algorithm. This implies that the size of the graph reduces to $2|\mathbf{B}| = 2^{O(s)}$. By applying the same approach as in Section 3.1, we obtain the following.

Theorem 4 *The schedulability problem for a sporadic constrained-deadline task system $\mathcal{T}$ can be solved in time $2^{O(s^2)}$ and space $O(s^2)$, where s is the input size of $\mathcal{T}$. Alternatively, it can be solved in time and space $2^{O(s)}$.*

Proof Any activation of Algorithm 2 needs $O(\log |\mathbf{B}|) = O(s)$ space, and the depth of the recursion is at most $O(\log |\mathbf{B}|) = O(s)$, so in total a space of $O(s^2)$

is enough. The running time can be found by the recurrence $T(k) = 2^{O(s)} \cdot 2 \cdot T(k/2) + O(1)$ which gives $T(k) = 2^{O(s \log k)}$ and finally $T(2|\mathbf{B}|) = 2^{O(s^2)}$.

The alternative complexity bound follows by using any linear-time reachability algorithm. □

Another consequence of the above construction is that if a given memoryless algorithm Alg is not optimal for a task system, then there is a job sequence on which Alg fails that has length $2^{O(s)}$.

4 Extension to the arbitrary-deadline model

In this section we drop the requirement that $D_i \leq P_i$ for all tasks $i \in [n]$. The techniques we used in the previous sections can still be applied; however, the definition of backlog configuration needs to be revised, and as a consequence some of the complexity bounds are increased. We limit ourselves to a discussion of the differences.

When $D_i > P_i$, it may happen that several jobs $j_1, \ldots, j_K$, generated from the same task i, are simultaneously awaiting completion. Our assumption, which is standard in this model [2], is that these jobs can only be processed serially (serial processing); more precisely, execution of $j_{k'}$ cannot begin before the completion of j_k, whenever $j_{k'}$ is released after j_k.

We notice that the definition of backlog configuration given in Section 2 is insufficient to capture the current state of the system. In particular, a single "remaining time to deadline" counter for each task is insufficient, because different jobs from the same task will have different absolute deadlines. Our first observation is a bound on the number of counters.

Proposition 1 *If at any time during the scheduling process there are more than $\lceil D_i/P_i \rceil$ pending jobs from task i, then some deadline has been missed.*

Proof If at time t there are $\lceil D_i/P_i \rceil + 1$ pending jobs from task i, since these jobs can be released at most once per P_i time units, the oldest job in the set has been released at least $P_i \cdot (\lceil D_i/P_i \rceil + 1) > D_i$ time units before t. But then the deadline of that job has been missed. □

Let $K_i := \lceil D_i/P_i \rceil + 1$. Observe that $K_i \leq D_i + 1$, since P_i is integral and positive. We define

$$\tilde{\mathbf{D}} = \times_{i=1}^{n} ([D_i] \cup \{0\})^{K_i}.$$

Remark 2 The size of $\tilde{\mathbf{D}}$ is $2^{2^{O(s)}}$, where s is the input size of $\mathcal{T}$.

The next observation concerns the "remaining execution time" counters. In principle, one would need K_i counters for the jobs of task i. However, because of the assumption that jobs from the same task need to be processed serially, it is enough to keep a single "remaining execution time" counter (the one for

the oldest job) together with a count of the *number* of pending jobs. Thus, we define

$$\tilde{\mathbf{C}} = \times_{i=1}^{n}([C_i] \cup \{0\}) \times ([K_i] \cup \{0\}).$$

Lastly, the "minimum remaining time to next activation" counters need not be modified, so we set $\tilde{\mathbf{P}} = \mathbf{P}$. The set of backlog configurations is now

$$\tilde{\mathbf{B}} := \tilde{\mathbf{C}} \times \tilde{\mathbf{D}} \times \tilde{\mathbf{P}}.$$

The notions of failure configuration, schedule, feasibility, etc. can now be defined as we did in Section 2. For the notion of schedule as a function $S : \mathbb{N} \to \mathbf{S}$, observe that we use once again the fact that jobs from the same task are processed serially, so it suffices to specify which m tasks have to be processed at any point in time.

The feasibility problem is solved in the same way as in Section 3.1, except that the graph is now based on the set

$$\tilde{\mathbf{B}}^{+} := \tilde{\mathbf{D}} \times \tilde{\mathbf{P}} \times 2^{\tilde{\mathbf{C}}}.$$

Since the set $\tilde{\mathbf{D}} \times \tilde{\mathbf{P}}$ completely captures the state of the environment, and the set $2^{\tilde{\mathbf{C}}}$ completely captures the state of all possible schedulers, it should be clear that by proceeding as in Section 3.1 and using $\tilde{\mathbf{B}}^{+}$ in place of $\mathbf{B}^{+}$ it is possible to define a graph for which Lemma 2 and Lemma 3 hold.

Remark 3 The size of both $\tilde{\mathbf{B}}$ and $\tilde{\mathbf{B}}^{+}$ is dominated by the size of $\tilde{\mathbf{D}}$, and so it is $2^{2^{O(s)}}$.

Analogously, the online feasibility problem and the schedulability problem are solved as before, except that the set $\tilde{\mathbf{B}}$ is used to construct the graph. After updating the bounds in Theorems 1–4 to reflect the size of the new sets $\tilde{\mathbf{B}}$ and $\tilde{\mathbf{B}}^{+}$, we obtain the following.

Theorem 5 *The feasibility problem, the online feasibility problem and the schedulability problem for a sporadic arbitrary-deadline task system $\mathcal{T}$ can all be solved in time $2^{2^{O(s)}}$, where s is the input size of $\mathcal{T}$. The feasibility problem and the schedulability problem can also be solved in space $2^{O(s)}$. Finally, if $\mathcal{T}$ is infeasible, there is a witness job sequence of length at most $2^{2^{O(s)}}$.*

Notice that we do not know whether the online feasibility problem can be solved in space $2^{O(s)}$ in this setting.

5 Continuous versus discrete schedules

In this section we show that, under our assumption of integer arrival times for the jobs, the feasibility of a sporadic task system does not depend on whether one is considering discrete or continuous schedules.

Let J be the (possibly infinite) set of jobs generated by a job sequence σ. In this section we do not need to keep track of which tasks generate the jobs,

so it will be convenient to use a somewhat different notation. Let r_j, c_j, $d_j \in \mathbb{N}$ denote respectively the release date, execution time and absolute deadline of a job j; so job j has to receive c_j units of processing in the interval $[r_j, d_j]$. A *continuous schedule* for J on m processors is a function $w : J \times \mathbb{N} \to \mathbb{R}_+$ such that:

1. $w(j,t) \leq 1$ for all $j \in J$ and $t \in \mathbb{N}$;
2. $\sum_{j \in J} w(j,t) \leq m$ for all $t \in \mathbb{N}$.

Quantity $w(j,t)$ is to be interpreted as the total amount of processing dedicated to job j during interval $[t, t+1]$. Thus, the first condition forbids the parallel execution of a job on more than one processor; the second condition limits the total volume processed in the interval by the m processors. The continuous schedule w is *feasible* for σ if it additionally satisfies

3. $\sum_{r_j \leq t < d_j} w(j,t) \geq c_j$ for all $j \in J$.

Finally, a task system $\mathcal{T}$ is *feasible with respect to continuous schedules* if any legal job sequence σ from $\mathcal{T}$ has a feasible continuous schedule. For the sake of clarity we call a system that is feasible in the sense defined in Section 2 *feasible with respect to discrete schedules*.

Lemma 5 *A finite job set J has a discrete schedule whenever it has a continuous schedule. This holds even when there are chain precedence constraints on J.*

Proof Consider first the case without precedence constraints. We setup an instance of a maximum flow problem whose solutions correspond to continuous schedules for σ, and whose integral solutions correspond to discrete schedules for σ; see also a similar construction in [4,5,8,15]. We build a network N consisting of four types of nodes:

1. a source node a;
2. for every $t = 0, 1, \ldots, \max_{j \in J} d_j$, a node x_t;
3. for every job j in σ, a node q_j;
4. a sink node z.

The arcs of the network are as follows:

1. from a to each Type 2 node, an arc with capacity m (m is the number of processors);
2. from each Type 2 node x_t to each Type 3 node q_j such that $r_j \leq t < d_j$, an arc with capacity 1;
3. from each Type 3 node q_j to z, an arc with capacity c_j.

Let K be the sum of the capacities of Type 3 edges.

Assume that a feasible continuous schedule w exists for σ. We now define a flow by setting the flow on each arc (a, x_t) to be $\sum_{j \in J} w(j,t)$; the flow on each arc (x_t, q_j) to $w(j,t)$; and the flow on each arc (q_t, z) to c_j. Now conditions (1) and (2) in the definition of continuous schedule for w ensure that the

capacity constraints are satisfied. Condition (3) ensures that the amount of flow entering any Type 3 node q_j is at least

$$\sum_{r_j \le t < d_j} w(j,t) \ge c_j.$$

Notice that some Type 3 node might have more flow entering the node than leaving it; but in that case we can still obtain a feasible flow of the same value by decreasing the incoming flow, and the capacities will not be violated.

Since all the capacities are integral we know there must be an integral flow attaining the same value as the optimal fractional flow; that is, value K. From this we can extract a discrete schedule by setting $w(j,t)$ equal to the flow on arc (x_t, q_j); this value must be either 0 or 1 by integrality. The total flow collected at each node q_j is exactly c_j. In this manner we have obtained a feasible discrete schedule for σ.

To extend the result to the case of chain precedence constraints on J, the construction above needs only a slight modification (a similar modification has been considered by Baptiste et al. [4] in a different context). Assume that job j precedes job j' and that, in the continuous schedule for σ, the completion time of j is after t, but the starting time of j' is before $t+1$. In this case j and j' are "competing" for the same time slot $[t, t+1)$, so instead of connecting q_j or $q_{j'}$ to x_t directly, we create a new node $v_{jj'}$ and three arcs $(x_t, v_{jj'})$, $(v_{jj'}, q_j)$, $(v_{jj'}, q_{j'})$ of unit capacity. In all other cases, that is, for time slots $[t, t+1)$ such that all of the following hold:

- t is greater or equal to the starting time of j in the continuous schedule (rounded down to an integer);
- t is strictly less than the completion time of j in the continuous schedule;
- j is not competing with another job in $[t, t+1)$;

we connect x_t to q_j by a unit capacity arc. The rest of the construction is similar to the previous one. It is straightforward to check that the existence of a continuous schedule for σ implies the existence of a feasible flow in the new network, so that, as before, the existence of an optimal integral flow implies the existence of a discrete schedule. □

Theorem 6 *A sporadic arbitrary-deadline task system $\mathcal{T}$ is feasible with respect to continuous schedules iff it is feasible with respect to discrete schedules.*

Proof If a task system is feasible with respect to discrete schedules, it is obviously also feasible with respect to continuous schedules: a discrete schedule is just a special case of a continuous schedule where $w(j,t) \in \{0,1\}$. So assume that a task system $\mathcal{T}$ is feasible with respect to continuous schedules, but not with respect to discrete schedules. Then there must be a witness job sequence σ that cannot be scheduled by any discrete schedule, but can be scheduled by some continuous schedule. By Theorem 5, we can assume that σ has some finite length $L = 2^{2^{O(s)}}$. So σ generates a *finite* collection of jobs J. But any feasible continuous schedule for a finite collection of jobs can be converted into

a feasible discrete schedule (Lemma 5). This holds even when chain precedence constraints are present, so that jobs generated by the same task are processed serially, as the model requires (recall the discussion in Section 4). The existence of a discrete schedule now contradicts the initial assumption that no such schedule existed. □

6 Concluding remarks

We have given upper bounds on the complexity of testing the feasibility, the online feasibility, and the schedulability of a sporadic task system on a set of identical processors. It is known that these three problems are at least coNP-hard [12]; however, no sharper hardness result is known. A natural question is to characterize more precisely the complexity of these problems, either by improving on the algorithms given here, or by showing that these problems are hard for some complexity class above coNP.

Given the apparent high complexity of feasibility testing, another natural direction is to investigate approximate feasibility tests, that trade off accuracy (in terms of the additional resources needed, such as processor speed) for computational efficiency [1,9]. In particular, the challenge here is to obtain tests that are more accurate, but still efficient, in the multiprocessor setting.

Acknowledgements We thank Sanjoy Baruah and Sebastian Stiller for useful discussions, and Nicole Megow for bringing reference [4] to our attention.

References

1. K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 187–195. IEEE, 2004.
2. T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In S. H. Son, I. Lee, and J. Y.-T. Leung, editors, *Handbook of Real-Time and Embedded Systems*, chapter 3. CRC Press, 2007.
3. T. P. Baker and M. Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In E. Tovar, P. Tsigas, and H. Fouchal, editors, *Proceedings of the 11th Conference on Principles of Distributed Systems*, volume 4878 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2007.
4. P. Baptiste, J. Carlier, A. Kononov, M. Queyranne, S. Sevastyanov, and M. Sviridenko. Integrality property in preemptive parallel machine scheduling. In A. E. Frid, A. Morozov, A. Rybalchenko, and K. W. Wagner, editors, *Proceedings of the 4th International Computer Science Symposium in Russia*, volume 5675 of *Lecture Notes in Computer Science*, pages 38–46. Springer, 2009.
5. S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.
6. S. K. Baruah and J. Goossens. Scheduling real-time tasks: Algorithms and complexity. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 28. CRC Press, 2003.
7. S. K. Baruah and K. Pruhs. Open problems in real-time scheduling. *Journal of Scheduling*, 13(6):577–582, 2009.

8. S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.
9. V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. A constant-approximate feasibility test for multiprocessor real-time scheduling. In D. Halperin and K. Mehlhorn, editors, *Proceedings of the 16th European Symposium on Algorithms*, volume 5193 of *Lecture Notes in Computer Science*, pages 210–221. Springer, 2008.
10. M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of the International Federation for Information Processing Congress*, pages 807–813. North-Holland, 1974.
11. R. Diestel. *Graph theory*. Springer, Heidelberg, 3rd edition, 2005.
12. F. Eisenbrand and T. Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In M. Charikar, editor, *Proceedings of the 21st Symposium on Discrete Algorithms*, pages 1029–1034. SIAM, 2010.
13. N. Fisher, J. Goossens, and S. K. Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1):26–71, 2010.
14. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.
15. W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.
16. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
17. R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.
18. J. H. Reif. The complexity of two-player games of incomplete information. *Journal of Computer and System Sciences*, 29(2):274–301, 1984.
19. W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and Systems Sciences*, 4(2):177–192, 1970.