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

Multiprocessor Schedulability of Arbitrary-Deadline Sporadic Tasks: Complexity and Antichain Algorithm

Gilles Geeraerts · Joël Goossens · Markus Lindström

Abstract Baker and Cirinei have introduced an exact but naive algorithm [3], that consists in solving a state *reachability* problem in a finite automaton, to check whether a set of sporadic hard real-time tasks is schedulable on an identical multiprocessor platform. However, this algorithm suffers from poor performance due to the exponential size of the automaton relative to the size of the task set. In this paper, we build on the work of Baker and Cirinei, and rely on their formalism to characterise the *complexity* of this problem. We prove that it is PSPACE-complete. In order to obtain an algorithm that is applicable in practice to systems of realistic sizes, we successfully apply techniques developed by the formal verification community, specifically *antichain techniques* [11] to this scheduling problem. For that purpose, we define and prove the correctness of a *simulation relation* on Baker and Cirinei's automaton. We show that our improved algorithm yields dramatically improved performance for the schedulability test and opens for many further improvements. This work is an extended and revised version of a previous conference paper by the same authors [21].

Keywords Hard real-time scheduling, formal methods, automata theory, sporadic tasks, exact schedulability test, multiprocessor scheduling, complexity.

1 Introduction

In this research we consider the schedulability problem of hard real-time, sporadic, arbitrary deadline and preemptive task systems upon identical multiprocessor platforms. Hard real-time systems are systems where tasks are not only required to provide correct computations but are also required to adhere to strict deadlines [22]. In the model we adopt, tasks are *sporadic*, which means that they are characterised by a *minimal inter-arrival time* T, which gives the *minimum* amount of time that should elapse between two successive job submissions by the task. In addition to this minimal inter-arrival time, tasks are also characterised by a *worst-case execution time* C, and an *arbitrary*

Université libre de Bruxelles
Département d'Informatique, Faculté des Sciences
Avenue Franklin D. Roosevelt 50, CP 212
1050 Bruxelles, Belgium
E-mail: {gilles.geeraerts, joel.goossens, mlindstr}@ulb.ac.be

deadline D (relative to submission of a job by the task). As the system is a *hard real-time system*, we request that, each time the scheduler captures a new job request in the system, at time t, the job is scheduled on a CPU during exactly C time units before time $t + D$.

Devising an *exact schedulability criterion* for sporadic task sets on multiprocessor platforms has so far proved difficult due to the fact that there is no known worst case scenario (i.e. critical instant). It was notably shown in [16] that the periodic case is not necessarily the worst on multiprocessor systems. In this context, the real-time community has mainly been focused on the development of *sufficient* schedulability tests that correctly identify all unschedulable task sets, but may misidentify some schedulable systems as being unschedulable [2] using a given platform and scheduling policy (see e.g. [5, 4]).

Baker and Cirinei have introduced the first algorithm [3] that checks *exactly* whether a sporadic task system is schedulable on an identical multiprocessor platform, using a given scheduling policy. In their setting, all the possible behaviours of the real-time system are represented by a finite (albeit large, see hereunder) automaton. Then, determining whether the task set is schedulable amounts to look for a so-called *failure state* that is reachable in this automaton from the initial state. Their approach to solve this reachability problem is a *brute-force* one, and consists in a traversal of the automaton, which may, in the worst case, visit all the states of the automaton. As the size of their automaton is, in the worst-case, exponential in the size of the task set, their algorithm does not perform well in practice, and does not always terminate within a reasonable delay even for small task sets with large enough periods [3].

In this paper, we first rely on Baker and Cirinei's formalisation of the scheduling problem to *characterise* precisely the *complexity* of checking whether a sporadic task set is schedulable under a given scheduler on an identical multiprocessor platform. We prove this problem to be PSPACE-complete. This high complexity might seem prohibitive, and might explain the relatively poor *practical* performance of Baker's and Cirinei's approach.

Although the complexity result might sound daunting, we show that it is possible to obtain better performance in practice, by applying techniques developed by the formal verification community, known as *antichain techniques*. These techniques have been introduced by Doyen *et al.* [11, 9] to solve reachability problems on finite automata. Their method behaves, in practice, better [11] than naive state traversal algorithms such as those used in [3].

An objective of this work is to be as self-contained as possible to allow readers from the real-time community to be able to fully understand the concepts borrowed from the formal verification community. We also hope our work will kickstart a "specialisation" of the methods presented herein within the realm of real-time scheduling, thus bridging the two communities in a novel fashion.

Related work. This work is not the first contribution applying techniques and models first proposed in the setting of formal verification to real-time scheduling. In the field of operational research, Abdeddaïm and Maler have studied the use of stopwatch automata to solve job-shop scheduling problems [1]. Cassez has recently exploited game theory, specifically timed games, to bound worst-case execution times on modern computer architectures, taking into account caching and pipelining [8]. Fersman *et al.* have studied a similar problem and introduced task automata which assume continuous time [12], whereas we consider discrete time in our work. They showed that, given

selected constraints, schedulability could be undecidable in their model. Bonifaci and Marchetti-Spaccamela have studied the related problem of feasibility of multiprocessor sporadic systems in [6] and have established an upper bound on its complexity. Guan *et al.* have used model-checking software tools to study the schedulability problem. In [18], the authors devise a timed automaton formulation of the multiprocessor schedulability problem on periodic tasks for fixed task priority policies. They implemented their automata using the Uppaal model-checking tool, which yielded an exact schedulability test. In [19], the authors used the NuSMV model-checking tool to devise an exact schedulability test for periodic tasks for fixed task priority and EDF policies.

This research. Our first contribution consists in revisiting and defining Baker and Cirinei's automaton in a more formal way than in [3]. This formal framework constitutes a strong theoretical basis for the results that we present afterwards. In order to show the relevance of this formalism, we also express various classical scheduling policy properties (such as memorylessness of the scheduler) in this framework.

Then, we study in detail the complexity of the scheduling problem of a sporadic task set on an identical multiprocessor platform, under a given scheduler. We show that this problem is PSpace-complete, by using a reduction from the universality problem for finite automata. The precise characterisation of the complexity gives us insights on the structure of this problem, and allows us to target specific heuristics that have proved to be efficient on problems with the same complexity. The heuristics we adopt here are the *antichain* techniques, that have been developed in the formal verification community.

Thus, our third contribution is to design a non-trivial *simulation relation* on the Baker-Cirinei automaton, and prove its correctness. This step is necessary to apply the *antichain techniques* to the analysis of the Baker and Cirinei automaton, and obtain a faster algorithm to detect the reachability of so-called *failure states* in this automaton. A failure state is a state in which a task has missed or will inevitably miss its deadline. Hence, when such a state is reachable in the automaton, the system is *not schedulable*.

To conclude, we show through implementation and experimental analysis that our proposed algorithm outperforms Baker and Cirinei's original brute-force algorithm.

Paper organisation. Section 2 defines the real-time scheduling problem we are focusing on, i.e. devising an exact schedulability test for sporadic task sets on identical multiprocessor platforms and formalise the model (a non-deterministic automaton) we will use to describe the problem and formulate how the schedulability test can be mapped to a reachability problem in this model. We also formalise various real-time scheduling concepts in the framework of our formal model.

Section 3 presents a reduction from the universality problem of finite automata to our schedulability problem, which proves that the latter is PSpace-complete. Section 4 then discusses how the reachability problem can be solved. We present the classical breadth-first algorithm used in [3] and we introduce an improved algorithm that makes use of techniques borrowed from the formal verification community [11]. The algorithm requires coarse *simulation relations* to work faster than the standard breadth-first algorithm. Section 5 introduces the *idle tasks simulation relation* which can be exploited by the aforementioned algorithm.

Section 6 then showcases experimental results comparing the breadth-first and our improved algorithm using the aforementioned simulation relation, showing that our algorithm outperforms the naive one. Section 7 concludes our work.

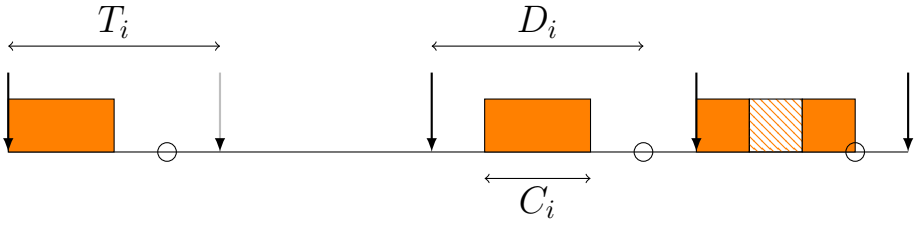


Fig. 1 Typical chronogram of a sporadic task. Such a task must wait at least T_i units of time between each request, each job must complete within D_i units of time after its arrival, and to do so must consume C_i units of processing time. A job can also be preempted by a job of another task (illustrated by the falling pattern).

2 Problem definition

We consider an identical multiprocessor platform with m processors and a sporadic task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Time is assumed to be discrete. A sporadic task τ_i is characterised by a *minimum interarrival time* $T_i > 0$, a *relative deadline* $D_i > 0$ and a *worst-case execution time* (also written WCET) $C_i > 0$. These parameters are illustrated in Fig. 1. A sporadic task τ_i submits a potentially infinite number of jobs to the system, with each request[1] being separated by at least T_i units of time. We will assume that tasks are preemptive and that jobs are not parallel, i.e. only execute on one single processor (though it may migrate from a processor to another during execution). We also assume jobs are independent, and that preemptions and migrations cost zero time. Since we consider arbitrary deadline systems, we know that several jobs of the same task can be active simultaneously. In this work, we assume that such jobs are considered sequentially, i.e. are served in a *first-in-first-out* (FIFO) manner. Note that this is not the only possible policy on multiprocessor platforms. Indeed, it has been shown in [15] that allowing task parallelism (i.e. allowing several jobs of a given arbitrary-deadline task to run simultaneously on different processors) is incomparable with the FIFO policy.

We wish to establish an *exact* schedulability test (i.e. a necessary and sufficient condition for schedulability) for any sporadic task set τ that tells us whether the set is schedulable on the platform with a given deterministic, sustainable (sometimes also called *predictable* in the literature), preemptive and FIFO scheduling policy.

As a reminder, the intuition behind a sustainable scheduling policy is that if a set of jobs consuming their worst-case execution time is schedulable, then that same set of jobs will necessarily be schedulable if any of the jobs requires less than its worst-case execution time [17].

Example 1 We will make use of a running example for the remainder of this paper. We define a sporadic arbitrary-deadline task system given in Table 1 hereunder, which we will simulate on a platform with $m = 2$ identical processors. $U_i = C_i/T_i$ denotes the system utilisation (or system load) of a given task, whereas $\delta_i = C_i/\min(D_i, T_i)$ denotes its density.

[1] Here, and in the rest of the paper, a 'request', means the moment at which the scheduler captures the request. In practice, the request could have been generated before, by an external event, but we abstract those low-level concerns away from our model.

	T_i	D_i	C_i	U_i	δ_i
τ_1	3	3	2	2/3	2/3
τ_2	3	3	2	2/3	2/3
τ_3	3	4	2	2/3	2/3

Table 1 The task set used as our running example.

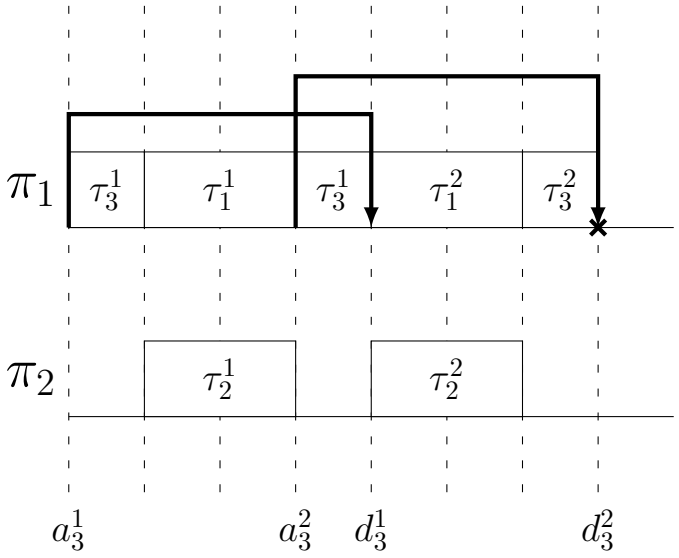


Fig. 2 Possible execution of the sporadic task system described in Example 1 on two processors π_1 and π_2 under global EDF scheduling. Arrows indicate arrival times and absolute deadlines of the first two jobs of τ_3. Preemptions are assumed to require negligible time. a_j^i indicates the arrival time of the ith job of task τ_j, and d_j^i its corresponding absolute deadline. τ_1 and τ_2 both make requests at instants $a_3^1 + 1$ and $a_3^2 + 1$.

Note that this system is not trivially schedulable (since there are more tasks than processors) and also not trivially unschedulable since the total system utilisation $U = \sum_i U_i = 2$ is no larger than the number m of processors. The total density $\delta = \sum_i \delta_i = 2$ is also no larger than m. Fig. 2 showcases a possible execution of this task set where τ_3 misses a deadline.

Automata-based semantics. To obtain a *formal definition* of the semantics of the real-time system, and of the schedulability problem, we rely on the automaton based formalism given by Baker and Cirinei [3]. Roughly speaking, Baker and Cirinei model all the possible executions of a task system by the possible paths of a finite automaton. Hence, determining whether an execution of the system leads to a deadline miss can be defined in terms of *reachability* of certain states in this automaton. This model allows use of *preemptive*, *deterministic* and *sustainable* scheduling policies. It can, however, be generalized to model broader classes of schedulers. We will discuss this aspect briefly in Section 7.

Definition 1 An *automaton* is a tuple $A = \langle V, E, S_0, F\rangle$, where V is a set of *states*, $E \subseteq V \times V$ is the set of *transitions*, $S_0 \subseteq V$ is a *set of initial states* and $F \subseteq V$ is a set of *target states*. An automaton is *finite* iff V is a finite set.

In the rest of the paper, we focus on the *reachability* problem (of target states) in finite automata. A *path* in an automaton $A = \langle V, E, S_0, F\rangle$ is a finite sequence $v_1, \ldots, v_\ell$ of states s.t. for all $1 \leqslant i \leqslant \ell - 1$: $(v_i, v_{i+1}) \in E$. Let $V' \subseteq V$ be a set of states of A. If there exists a path $v_1, \ldots, v_\ell$ in A s.t. $v_\ell \in V'$, we say that v_1 *can reach* V'. For an automaton A, we denote by $\mathsf{Reach}(A)$ the set of states that are reachable from an initial state of A:

Definition 2 (Reachable states) The set of reachable states of an automaton A is

$$\mathsf{Reach}(A) \stackrel{\mathbf{def}}{=} \{v \in V \mid \exists \text{ a path } v_1, \ldots, v_\ell : v_1 \in S_0 \wedge v_\ell = v\}$$

Then, the *reachability problem* asks, given an automaton A whether the set of target states F is reachable in A, i.e. whether $\mathsf{Reach}(A) \cap F \neq \emptyset$.

Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of sporadic tasks and m be a number of processors. This section is devoted to explaining how to model the behaviour of such a system by means of an automaton A, and how to reduce the schedulability problem of τ on m processors to an instance of the reachability problem in A. At any moment during the execution of such a system, the information we need to retain about each task τ_i are: (i) the *earliest next arrival time* $\mathrm{nat}(\tau_i)$ relative to the current instant and (ii) the remaining processing time $\mathrm{rct}(\tau_i)$ of the currently ready job of τ_i. As an example, consider Fig. 2 again. At the first instant on the figure, $\mathrm{nat}(\tau_3) = 3$ and $\mathrm{rct}(\tau_3) = 2$. At the third instant, $\mathrm{nat}(\tau_3) = 1$ and $\mathrm{rct}(\tau_3) = 1$ too, as job τ_3^1 still needs on time unit to complete. Let us build on this intuition to define the notion of *system states*:

Definition 3 (System states) Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of sporadic tasks. A *system state* of τ is a tuple $S = \langle \mathrm{nat}_S, \mathrm{rct}_S\rangle$ where

- nat_S is a function from τ to $\mathbb{N}$ s.t. for all τ_i: $\mathrm{nat}_S(\tau_i) \leqslant T_{\max}$, with $T_{\max} \stackrel{\mathbf{def}}{=} \max_i T_i$, and
- rct_S is a function from τ to $\{0, 1, \ldots, C_{\max}\}$, with $C_{\max} \stackrel{\mathbf{def}}{=} \max_i C_i$.

We denote by $\mathsf{States}(\tau)$ the set of all system states of τ.

Remark that, for each task set τ, the set $\mathsf{States}(\tau)$ is *infinite*. However, the only source of infinity stems from the fact that we provide no lower bound on $\mathrm{nat}_S(\tau_i)$ (all the $\mathrm{rct}_S(\tau_i)$ are clearly within a bounded interval. Recall that $\mathrm{rct}_S(\tau_i)$ models the number of computation time unit that are necessary for the job to complete, which is a positive value, bounded above by C_i). We will show, at the end of the present section, that we can actually consider a lower bound on nat, and thus restrict ourselves to a finite subset of $\mathsf{States}(\tau)$.

In order to define the set of transitions of the automaton, we need to rely on ancillary notions:

Definition 4 (Eligible task) A task τ_i is *eligible* in the state S if it can submit a job (i.e. if and only if the task does not currently have an active job and the last job was submitted at least T_i time units ago) from this configuration. Formally, the set of eligible tasks in state S is:

$$\mathsf{Eligible}(S) \stackrel{\mathbf{def}}{=} \{\tau_i \mid \mathrm{nat}_S(\tau_i) \leqslant 0 \wedge \mathrm{rct}_S(\tau_i) = 0\}$$

Definition 5 (Active task) A task is *active* in state S if it currently has a job that has not finished in S. Formally, the set of active tasks in S is:

$$\mathsf{Active}(S) \stackrel{\mathbf{def}}{=} \{\tau_i \mid \mathrm{rct}_S(\tau_i) > 0\}$$

A task that is not active in S is said to be *idle* in S. We can define the set of idle tasks as the complement of the set of active tasks:

$$\mathsf{Idle}(S) \stackrel{\mathbf{def}}{=} \tau \setminus \mathsf{Active}(S)$$

Definition 6 (Laxity [3]) The laxity of a task τ_i in a system state S is:

$$\mathrm{laxity}_S(\tau_i) \stackrel{\mathbf{def}}{=} \mathrm{nat}_S(\tau_i) - (T_i - D_i) - \mathrm{rct}_S(\tau_i)$$

Remark that, when $T_i = D_i$ for all task τ_i (that is, we consider a system with *implicit deadlines*), the definition of laxity becomes: $\mathrm{laxity}_S(\tau_i) \stackrel{\mathbf{def}}{=} \mathrm{nat}_S(\tau_i) - \mathrm{rct}_S(\tau_i)$.

The laxity of a task is thus the difference between the number of processing time units that the current job of the task needs to complete, and the number of time units before its deadline. Thus the laxity can be regarded as the maximum number of time units that the current job can idle before its next deadline, while still ensuring that it will not miss its deadline. Obviously, if the laxity becomes < 0, the current job of the task will inevitably miss its deadline: as the job cannot be parallelised, it will not be allotted a sufficient amount of CPU time before its deadline. This motivates the definition of *failure state*:

Definition 7 (Failure state) A state S is a *failure state* iff the laxity of at least one task is negative in S. Formally, the set of failure states on τ is:

$$\mathsf{Fail}_\tau \stackrel{\mathbf{def}}{=} \{S \mid \exists \tau_i \in \tau : \mathrm{laxity}_S(\tau_i) < 0\}$$

Thanks to these notions we are now ready to explain how to build the transition relation of the automaton that models the behaviour of τ. For that purpose, we first choose a *scheduler*. Intuitively, a scheduler is a *function*[2] Run that maps each state S to a set of at most m active tasks $\mathsf{Run}(S)$ to be run:

Definition 8 (Scheduler) A (deterministic) *scheduler* for τ on m processors is a function $\mathsf{Run} : \mathsf{States}(\tau) \to 2^\tau$ s.t. for all S: $\mathsf{Run}(S) \subseteq \mathsf{Active}(S)$ and $0 \leqslant |\mathsf{Run}(S)| \leqslant m$. Moreover:

1. Run is *work-conserving* iff for all S, $|\mathsf{Run}(S)| = \min\{m, |\mathsf{Active}(S)|\}$
2. Run is *memoryless* iff for all $S_1, S_2 \in \mathsf{States}(\tau)$ with $\mathsf{Active}(S_1) = \mathsf{Active}(S_2)$:

$$\forall \tau_i \in \mathsf{Active}(S_1) : \begin{pmatrix} \mathrm{nat}_{S_1}(\tau_i) = \mathrm{nat}_{S_2}(\tau_i) \\ \wedge\, \mathrm{rct}_{S_1}(\tau_i) = \mathrm{rct}_{S_2}(\tau_i) \end{pmatrix} \text{ implies } \mathsf{Run}(S_1) = \mathsf{Run}(S_2)$$

Intuitively, the work-conserving property implies that the scheduler always exploits as many processors as available. The memoryless property implies that the decisions of the scheduler are not affected by tasks that are idle and that the scheduler does not consider the past to make its decisions.

As examples, we can formally define the preemptive global *deadline monotonic* (DM) and *earliest deadline first* (EDF) schedulers [22].

[2] Remark that by modeling the scheduler as a function, we restrict ourselves to *deterministic schedulers*.

Definition 9 (Preemptive global DM scheduler) Let $\ell \stackrel{\textbf{def}}{=} \min\{m, |\mathsf{Active}(S)|\}$. Then, $\mathsf{Run}_{\mathrm{DM}}$ is a function that computes $\mathsf{Run}_{\mathrm{DM}}(S) \stackrel{\textbf{def}}{=} \{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_\ell}\}$ s.t. for all $1 \leqslant j \leqslant \ell$ and for all τ_k in $\mathsf{Active}(S) \setminus \mathsf{Run}_{\mathrm{DM}}(S)$, we have $D_k > D_{i_j}$ or $D_k = D_{i_j} \wedge k > i_j$.

Intuitively, (preemptive global) DM orders the tasks according to their *relative* deadlines (and then according to their index, when two deadlines are equal), assigning the highest priority to the tasks with the smallest relative deadlines. When scheduling the tasks on m CPUs, DM selects the m most higher priority active tasks (if they exist), according to this priority assignment.

Definition 10 (Preemptive global EDF scheduler) Let $\mathrm{ttd}_S(\tau_i) \stackrel{\textbf{def}}{=} \mathrm{nat}_S(\tau_i) - (T_i - D_i)$ be the time remaining before the *absolute* deadline of the last submitted job [3] of $\tau_i \in \mathsf{Active}(S)$ in state S. Let $\ell \stackrel{\textbf{def}}{=} \min\{m, |\mathsf{Active}(S)|\}$. Then, $\mathsf{Run}_{\mathrm{EDF}}$ is a function that computes $\mathsf{Run}_{\mathrm{EDF}}(S) \stackrel{\textbf{def}}{=} \{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_\ell}\}$ s.t. for all $1 \leqslant j \leqslant \ell$ and for all τ_k in $\mathsf{Active}(S) \setminus \mathsf{Run}_{\mathrm{EDF}}(S)$, we have $\mathrm{ttd}_S(\tau_k) > \mathrm{ttd}_S(\tau_{i_j})$ or $\mathrm{ttd}_S(\tau_k) = \mathrm{ttd}_S(\tau_{i_j}) \wedge k > i_j$.

Intuitively, (preemptive global) EDF orders the active jobs according to the distance to their next *absolute* deadline, assigning the highest priority to the jobs that the closest to their *absolute* deadlines (and using the index of the tasks to break ties). When scheduling a set of active jobs on m CPUs, EDF selects the m highest priority active tasks (if they exist), according to this priority assignment.

By Definition 8, global DM and EDF are thus work-conserving and it can also be verified that they are memoryless.

Notice that Definition 8 (scheduler) is not only designed to formalize priority-driven and preemptive schedulers but can be used to consider non-preemptive or partitioned schedulers. Non-preemptive schedulers are easy to formalize since the function Run depends on the tuple $< \mathrm{nat}_S(\tau_i), \mathrm{rct}_S(\tau_i) >$, e.g. if $\mathrm{rct}_S(\tau_i) < C_i$ a non-preemptive scheduler *must* put τ_i in the set Run. In the same vein it is easy to formalize a partitioned scheduler as well.

In [3], suggestions to model several other schedulers were presented. It was particularly shown that adding supplementary information to system states could allow broader classes of schedulers to be used. Intuitively, states could e.g. keep track of what tasks were executed in their predecessor to implement non-preemptive schedulers.

Clearly, in the case of the scheduling of sporadic tasks, two types of events can modify the current state of the system:

1. *Clock-tick transitions* model the elapsing of time for one time unit, i.e. the execution of the scheduler and the running of jobs.
2. *Request transitions* (called *ready transitions* in [3]) model requests[3] from sporadic tasks at a given instant in time.

Let S be a state in $\mathsf{States}\,(\tau)$, and let Run be a scheduler. Then, letting one time unit elapse from S under the scheduling policy imposed by Run amounts to decrementing the rct of the tasks in $\mathsf{Run}(S)$ (and only those tasks), and to decrementing the nat of all tasks. Formally:

[3] Recall that this is actually the moment at which the scheduler captures the request.

Definition 11 Let $S = \langle \mathrm{nat}_S, \mathrm{rct}_S \rangle \in \mathsf{States}(\tau)$ be a system state and Run be a scheduler for τ on m processors. Then, we say that $S^+ = \langle \mathrm{nat}_S^+, \mathrm{rct}_S^+ \rangle$ is a *clock-tick successor* of S under Run, denoted $S \xrightarrow{\mathsf{Run}} S^+$ iff:

1. for all $\tau_i \in \mathsf{Run}(S)$: $\mathrm{rct}_S^+(\tau_i) = \mathrm{rct}_S(\tau_i) - 1$;
2. for all $\tau_i \notin \mathsf{Run}(S)$: $\mathrm{rct}_S^+(\tau_i) = \mathrm{rct}_S(\tau_i)$;
3. for all $\tau_i \in \tau$: $\mathrm{nat}_S^+(\tau_i) = \begin{cases} \max\{\mathrm{nat}_S(\tau_i) - 1, 0\} & \text{if } \mathrm{rct}_S(\tau_i) = 0 \\ \mathrm{nat}_S(\tau_i) - 1 & \text{if } \mathrm{rct}_S(\tau_i) > 0 \end{cases}$.

Let S be a state in $\mathsf{States}(\tau)$. Intuitively, when the system is in state S, a request by some task τ_i for submitting a new job has the effect to update S by setting $\mathrm{nat}(\tau_i)$ to T_i and $\mathrm{rct}(\tau_i)$ to C_i. This can be generalised to sets of tasks. Formally:

Definition 12 Let $S \in \mathsf{States}(\tau)$ be a system state and let $\tau' \subseteq \mathsf{Eligible}(S)$ be a set of tasks that are eligible to submit a new job in the system. Then, we say that S' is a *τ'-request successor* of S, denoted $S \xrightarrow{\tau'} S'$, iff:

1. for all $\tau_i \in \tau'$: $\mathrm{nat}_S(\tau_i) + T_i \leqslant \mathrm{nat}_{S'}(\tau_i) \leqslant T_i$ and $\mathrm{rct}_{S'}(\tau_i) = C_i$
2. for all $\tau_i \in \tau \setminus \tau'$: $\mathrm{nat}_{S'}(\tau_i) = \mathrm{nat}_S(\tau_i)$ and $\mathrm{rct}_{S'}(\tau_i) = \mathrm{rct}_S(\tau_i)$.

For a given state S, we'll let $S^{\tau'}$ denote the set of its τ'-request successors.

Remark that we allow $\tau' = \emptyset$ (that is, no task asks to submit a new job in the system). Also note that each system state has a *set* of possible τ'-request successors for a given τ' due to the possibility of choosing $\mathrm{nat}_{S'}(\tau_i)$ among a set of values. Finally, also remark that since we only consider *sustainable* scheduling policies, we can limit ourselves to considering cases where the remaining processing time of a new job is equal to its worst-case execution time.

We are now ready to define the automaton $\overline{A}(\tau, \mathsf{Run})$ that formalises the behavior of the system of sporadic tasks τ, when executed upon m processors under a scheduling policy Run:

Definition 13 Given a set of sporadic tasks τ and a scheduler Run for τ on m processors, the automaton $\overline{A}(\tau, \mathsf{Run})$ is the tuple $\langle V, E, S_0, F \rangle$ where:

1. $V = \mathsf{States}(\tau)$
2. $(S_1, S_2) \in E$ iff there exists an *intermediate state* $S' \in \mathsf{States}(\tau)$ and $\tau' \subseteq \tau$ s.t. $S_1 \xrightarrow{\tau'} S' \xrightarrow{\mathsf{Run}} S_2$.
3. $S_0 = \{\langle \mathrm{nat}_0, \mathrm{rct}_0 \rangle\}$ where for all $\tau_i \in \tau$, $\mathrm{nat}_0(\tau_i) = \mathrm{rct}_0(\tau_i) = 0$.
4. $F = \mathsf{Fail}_\tau$

Remark that, according to this definition, whenever $(S_1, S_2) \in E$, there exists an intermediate state S' that is reached from S_1 after some eligible tasks have submitted a job, but before the clock tick. However, these *intermediate states* are *not reachable*, as they are somehow *hidden* in the transition (S_1, S_2). However, this is not a problem as the reachable states are sufficient to detect the deadline misses. We have introduced this notion of intermediate state for the sake of clarity as it allows us to separate *clock ticks* transition and *requests transitions*.

We remark that our definition deviates slightly from that of Baker and Cirinei. In our definition, a path in the automaton corresponds to an execution of the system that alternates between requests transitions (possibly with an empty set of requests)

and clock-tick transitions. In their work [3], Baker and Cirinei allow any sequence of clock ticks and requests, but restrict each request to a single task at a time. It is easy to see that these two definitions are equivalent. A sequence of k clock ticks in Baker's automaton corresponds in our case to a path $S_1, S_2, \dots S_{k+1}$ s.t. for all $1 \leqslant i \leqslant k$: $S_i \xrightarrow{\emptyset} S_i \xrightarrow{\mathsf{Run}} S_{i+1}$. A maximal sequence of successive requests by $\tau_1, \tau_2, \dots, \tau_k$, followed by a clock tick corresponds in our case to a single edge (S_1, S_2) s.t. $S_1 \xrightarrow{\{\tau_1,\dots,\tau_k\}} S' \xrightarrow{\mathsf{Run}} S_2$ for some S'. Conversely, each edge (S_1, S_2) in $A(\tau, \mathsf{Run})$ s.t. $S_1 \xrightarrow{\tau'} S' \xrightarrow{\mathsf{Run}} S_2$, for some state S' and set of tasks $\tau' = \{\tau_1, \dots, \tau_k\}$, corresponds to a sequence of successive requests[4] by $\tau_1, \dots, \tau_k$ followed by a clock tick in Baker's and Cirinei's setting.

Multiprocessor Sporadic Schedulability. The purpose of the definition of $\overline{A}(\tau, \mathsf{Run})$ should now be clear to the reader. Each possible execution of the system corresponds to a path in $\overline{A}(\tau, \mathsf{Run})$ and vice-versa. States in Fail_τ correspond to states of the system where a deadline will unavoidably be missed. Thus, for the system to be schedulable, such failure states should not be reachable in the automaton. Otherwise, this would mean that there exists a possible activation sequence of the tasks that leads to a deadline miss. Hence, we can define the problem of determining whether a given set of sporadic tasks is schedulable under a given scheduler in terms of the reachability of failure states in the automaton:

Problem 1 Given a sporadic task set τ and a scheduler Run on m processors, the *multiprocessor sporadic schedulability (*MSS*) problem* asks whether Fail_τ is reachable in $\overline{A}(\tau, \mathsf{Run})$ or not.

When the answer to this problem is negative, we say that τ *is schedulable under* Run.

Baker-Cirinei automaton. Unfortunately, according to definition 13, $\mathsf{Reach}(\overline{A}(\tau, \mathsf{Run}))$ is potentially *infinite* as $\mathsf{States}(\tau)$ is infinite, and the definition of the transition relation allows for infinite path visiting infinitely many different states. For instance, consider a system with a single task τ_1 with $C_1 = 1$, $T_1 = 1$ and $D_1 = 1$, and a scheduler Run s.t. $\mathsf{Run}(S) = \emptyset$ for all S. Then, consider the infinite run $S_0, S_1, S_2, \dots, S_i, \dots$ of $A(\{\tau_1\}, \mathsf{Run})$ s.t.:

$$S_0 \xrightarrow{\{\tau_1\}} S'_0 \xrightarrow{\mathsf{Run}} S_1 \xrightarrow{\emptyset} S'_1 \xrightarrow{\mathsf{Run}} S_2 \xrightarrow{\emptyset} S'_2 \xrightarrow{\mathsf{Run}} S_3 \cdots S_i \xrightarrow{\emptyset} S'_i \xrightarrow{\mathsf{Run}} S_{i+1} \cdots$$

It is easy to check that, for all $i \geqslant 1$: $\mathrm{rct}_{S_i}(\tau_1) = 1$ and $\mathrm{nat}_{S_i}(\tau_i) = 1 - i$. Hence, all states in this run are pairwise different and $\mathsf{Reach}(\overline{A}(\tau, \mathsf{Run}))$ is infinite.

Intuitively, to detect a deadline miss, it is sufficient to reach a state S where $\mathrm{laxity}_S(\tau_i) = -1$ for some task τ_i. Indeed, if such a state exists, then τ_i has either missed a deadline or will inevitably miss it. Moreover, each state S' s.t. $\mathrm{laxity}_{S'}(\tau_i) < -1$ has necessarily been obtained by traversing first a state where $\mathrm{laxity}_S(\tau_i) = -1$. Thus, we can restrict ourselves to states where $\mathrm{laxity}_S(\tau_i) \geqslant -1$. As the laxity is also always bounded above, this reduces the set of states that we need to explore to a *finite* set. This crucial observation was already made by Baker and Cirinei [3].

Definition 14 Let τ be a set of sporadic tasks, let Run be a scheduler for τ on m processors, and let $\overline{A}(\tau, \mathsf{Run}) = \langle V, E, V_0, \mathsf{Fail}_\tau \rangle$ be the associated automaton (see Definition 13). Then, the *Baker-Cirinei* automaton associated to τ and Run is the automaton $A(\tau, \mathsf{Run}) = \langle V', E', V'_0, F' \rangle$ where:

[4] Remark that the order does not matter.

– $V' = \{S \in \mathsf{States}\,(\tau) \mid \forall \tau_i \in \tau = \mathrm{laxity}_S(\tau_i) \geqslant -1\}$,
– $E' = E \cap (V' \times V')$,
– $V'_0 = V_0 \cap V'$,
– $F' = \mathsf{Fail}_\tau \cap V'$.

That is, $A(\tau, \mathsf{Run})$ is the restriction of $\overline{A}(\tau, \mathsf{Run})$ to the states where all tasks have a laxity larger than or equal to -1. Since $A(\tau, \mathsf{Run})$ is a restriction of $\overline{A}(\tau, \mathsf{Run})$, all run of the former that reach a failure state will also be present in the latter. However, we can also show that $A(\tau, \mathsf{Run})$ is *sufficient* to detect states failure states:

Proposition 1 *For all set of sporadic task τ and all scheduler* Run *for τ: Fail_τ is reachable in $A(\tau, \mathsf{Run})$ iff F' is reachable in $\overline{A}(\tau, \mathsf{Run})$.*

Proof Let us assume $\overline{A}(\tau, \mathsf{Run}) = \langle V, E, V_0, \mathsf{Fail}_\tau \rangle$ and $A(\tau, \mathsf{Run}) = \langle V', E', V'_0, \mathsf{Fail}_\tau \rangle$. As all runs of $A(\tau, \mathsf{Run})$ are runs of $\overline{A}(\tau, \mathsf{Run})$, by definition, $\mathsf{Reach}(A(\tau, \mathsf{Run})) \subseteq \mathsf{Reach}(\overline{A}(\tau, \mathsf{Run}))$. Moreover, since $F' \subseteq \mathsf{Fail}_\tau$, we conclude that $F' \cap \mathsf{Reach}(A(\tau, \mathsf{Run})) \neq \emptyset$ implies $\mathsf{Fail}_\tau \cap \mathsf{Reach}(\overline{A}(\tau, \mathsf{Run})) \neq \emptyset$.

For the reverse direction, we assume that $S_0, S_1, \ldots, S_\ell$ is a run of $\overline{A}(\tau, \mathsf{Run})$ s.t. $S_\ell \in \mathsf{Fail}_\tau$. Let k be the least position s.t. $k \leqslant \ell$, $S_k \in \mathsf{Fail}_\tau$, and for all $k \leqslant j \leqslant \ell$: $S_j \in \mathsf{Fail}_\tau$. That is $S_k, S_{k+1}, \ldots, S_\ell$ is the longest suffix of S_ℓ that visits only failure states. W.l.o.g., we can assume that for all $0 \leqslant i < k$: $S_i \notin \mathsf{Fail}_\tau$, i.e., we have not traversed failure states before the S_k, S_{k+1}, S_ℓ suffix. We also assume a task τ_n s.t. $\mathrm{laxity}_{S_i}(\tau_n) < 0$ for all $k \leqslant i \leqslant \ell$. Thus, for all $k \leqslant i \leqslant \ell$, $\mathrm{rct}_{S_i}(\tau_n) > 0$. As a consequence, τ_n has never completed a job along the $S_k, \ldots, S_\ell$ suffix, and has thus never submitted a new job along this suffix. Hence, by definition of the automaton, the laxity of τ_n has *decreased* along the suffix $S_k, \ldots, S_\ell$. Since $S_{k-1} \notin \mathsf{Fail}_\tau$, but $S_k \in \mathsf{Fail}_\tau$, we conclude that $\mathrm{laxity}_{S_k}(\tau_n) = -1$, and that $\mathrm{laxity}_{S_k}(\tau_m) \geqslant -1$ for all $m \neq n$, hence $S_k \in F'$. As the laxity is positive for all tasks in all states traversed along the prefix that reaches S_k, by hypothesis, we conclude that $S_0, \ldots, S_k$ is a run of $A(\tau, \mathsf{Run})$. Hence F' is reachable in $A(\tau, \mathsf{Run})$. □

Remark that, since $\mathrm{laxity}_S(\tau_i) \geqslant -1$ in all states S of $A(\tau, \mathsf{Run})$, and for all task τ_i, we can deduce that $\mathrm{nat}_S(\tau_i) \geqslant T_i - D_i$ in all those states, and for all τ_i:

Proposition 2 *For all $S \in \mathsf{Reach}(A(\tau, \mathsf{Run}))$, for all task τ_i: $\mathrm{nat}_S(\tau_i) \geqslant T_i - D_i$.*

Proof First recall that, for all state S, and all task τ_i, $\mathrm{rct}_S(\tau_i) \geqslant 0$. Moreover, for all state S and all task τ_i: $\mathrm{laxity}_S(\tau_i) = -1$ *implies that* $\mathrm{rct}_S(\tau_i) \geqslant 1$, by definition of the laxity. Let S be a state in $\mathsf{Reach}(A(\tau, \mathsf{Run}))$ and let τ_i be a task. We consider two cases.

1. If $\mathrm{laxity}_S(\tau_i) \geqslant 0$, then:

$$\begin{array}{lrl} & \mathrm{laxity}_S(\tau_i) \geqslant 0 & \\ \Leftrightarrow & \mathrm{nat}_S(\tau_i) - T_i + D_i - \mathrm{rct}_S(\tau_i) \geqslant 0 & \text{Def. of laxity} \\ \Leftrightarrow & \mathrm{nat}_S(\tau_i) \geqslant \mathrm{rct}_S(\tau_i) + T_i - D_i & \\ \Rightarrow & \mathrm{nat}_S(\tau_i) \geqslant T_i - D_i & \mathrm{rct}_S(\tau_i) \geqslant 0 \end{array}$$

2. If $\mathrm{laxity}_S(\tau_i) = -1$, then:

$$\begin{array}{lrl} & \mathrm{laxity}_S(\tau_i) = -1 & \\ \Leftrightarrow & \mathrm{nat}_S(\tau_i) - T_i + D_i - \mathrm{rct}_S(\tau_i) = -1 & \text{Def. of laxity} \\ \Leftrightarrow & \mathrm{nat}_S(\tau_i) = \mathrm{rct}_S(\tau_i) + T_i - D_i - 1 & \\ \Rightarrow & \mathrm{nat}_S(\tau_i) \geqslant 1 + T_i - D_i - 1 & 1 \leqslant \mathrm{rct}_S(\tau_i) \\ \Leftrightarrow & \mathrm{nat}_S(\tau_i) \geqslant T_i - D_i & \end{array}$$

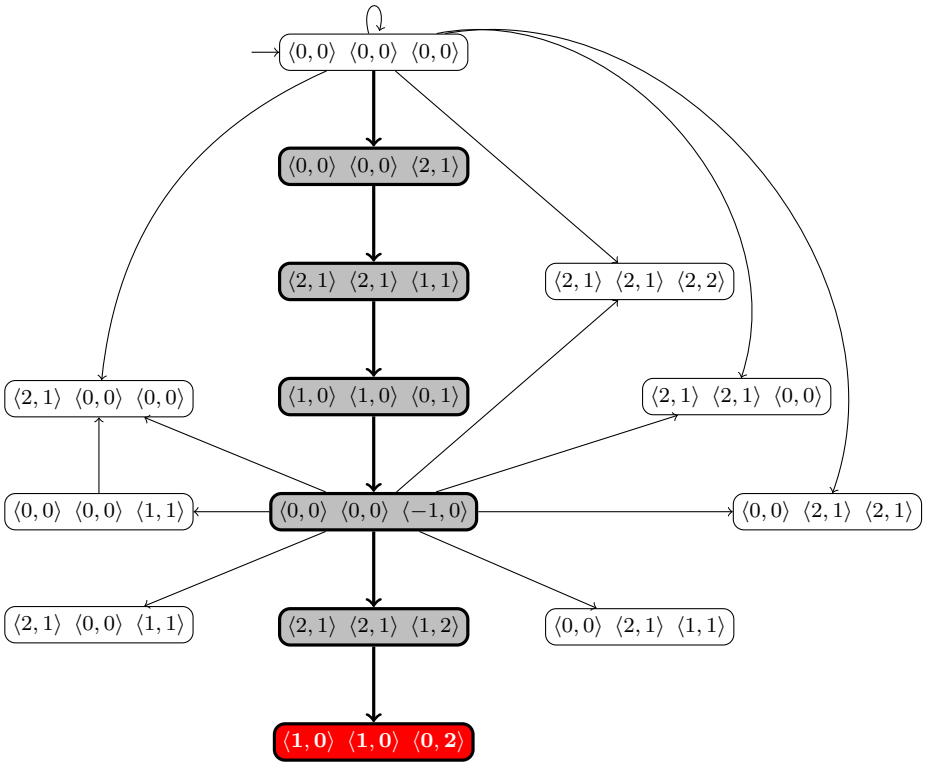


Fig. 3 Parts of the automaton described in Example 2. The highlighted path corresponds to the scenario played out in Fig. 2 (see p. 5).

In both cases, we conclude that $\mathrm{nat}_S(\tau_i) \geqslant T_i - D_i$. □

This result thus establishes that $A(\tau, \mathsf{Run})$ is indeed a *finite* automaton, as both values $\mathrm{rct}_S(\tau_i)$ and $\mathrm{nat}_S(\tau_i)$ take their values in a finite interval, for all state S and all task τ_i. Moreover, Proposition 1 tells us that the real-time system will miss a deadline iff a failure state is reachable in $A(\tau, run)$. Thus, MSS is trivially decidable, as it is sufficient to explore all the (finitely many) reachable states of $A(\tau, run)$ to determine whether a failure state is reachable or not. This is essentially the algorithm presented by Baker and Cirinei [3]. In the sequel, we will base all our reasoning on this automaton, and we will sometimes abuse notations by implicitly assuming that, for all states $S \in \mathsf{States}(\tau)$, for all task τ_i: $\mathrm{nat}_S(\tau_i) \in [T_{\min} - D_{\max}, T_{max}]$, where $T_{\min} \stackrel{\mathbf{def}}{=} \min_i T_i$, $T_{\max} \stackrel{\mathbf{def}}{=} \max_i T_i$ and $D_{\max} \stackrel{\mathbf{def}}{=} \max_i D_i$.

Example 2 Fig. 3 illustrates a possible graphical representation of a (partial) Baker-Cirinei automaton. On this example, the automaton depicts our previous running example (see Table 1 on p. 5), an EDF-schedulable sporadic task set using an EDF scheduler and assuming $m = 2$.

System states are represented by nodes. We represent a system state S with the $\langle \mathrm{nat}(\tau_i), \mathrm{rct}(\tau_i) \rangle$ format, with $i \in \{1, 2, 3\}$. Edges between states model the transitions of our automaton. For space purposes, we deliberately truncated the automaton to

showcase important features: the initial state S_0 at the top, transitions from a state where a nat is negative, and a failure state.

We have elected to highlight a specific path in the automaton, corresponding to the scenario that played out in Fig. 2 (see p. 5). At the first instant ($t = 0$), our automaton is in its initial state where all tasks are eligible. We then assume τ_3 makes a request at $t = 0$, and this job gets scheduled by EDF. At time $t = 1$, τ_1 and τ_2 both make a request. These two jobs end at $t = 3$, at which point the first job of τ_3 gets scheduled and ends its run. Note that in the automaton, following a transition from one state to another always corresponds to moving ahead one unit of time in the actual real-time system we model.

The automaton reaches the state $(\langle 0, 0\rangle, \langle 0, 0\rangle, \langle -1, 0\rangle)$ which is of interest because of the appearance of a negative value for a $\mathrm{nat}(\tau_3)$. The semantics of a negative value of a nat is the following in terms of the actual real-time system: let a_i^k be the arrival time of the current (or last) job, number k, of τ_i. A negative nat reflects the fact that the current instant (in terms of the real-time system we model) is $a_i^k + T_i - \mathrm{nat}$, i.e. that $|\,\mathrm{nat}\,|$ units of time have elapsed since the minimum arrival time of the next job. In practice, this means that, as we reach a state where nat is negative and rct is zero, the successors of that state account for all possibilities of arrival for the next job of the task in question. For instance, consider all the successors of the state S where $\mathrm{nat}(\tau_3) = -1$ in Fig. 3, and assume that state S corresponds to instant t in the real-time system. The different possible states for τ_3 in the successors of S (thus at instant $t+1$) are the following. First, $\langle 0, 0\rangle$ which means that τ_3 has not submitted another job by instant t. Second, either $\langle 2, 1\rangle$ or $\langle 2, 2\rangle$, meaning that τ_3 has scheduled a job at instant t (and has been scheduled in the case of $\langle 2, 1\rangle$ or not in the case of $\langle 2, 2\rangle$), hence τ_3 must wait another $\mathrm{nat}(\tau_3) = 2$ time units (that is, instant $t + 3$) before submitting a fresh job. Finally, $\langle 1, 1\rangle$ which means that τ_3 *has submitted a job at instant* $t - 1$, and that this job has been scheduled (at instant t). Hence, since τ_3 has submitted at $t - 1$, and $T_3 = 3$, it has to wait up to $t+2$ before submitting a new job. Thus, there is a possible successor $S'$of S with $\mathrm{nat}_{S'}(\tau_3) = 1$.

Unfortunately, albeit $A(\tau, \mathsf{Run})$ is finite, the size of its state space can be intractable even for very small sets of tasks τ. This difficulty is formalised in the next section, where we prove that this problem is in fact PSPACE-complete.

3 Complexity

In this section, we address the computational complexity of MSS and prove that it is PSPACE-complete. Recall [24] that the class PSPACE contains all problems that can be recognised by a Turing machine using polynomial space. That is, for all problem in PSPACE, there exists a Turing machine M and a polynomial $p(n)$ s.t., along all runs of M on an input of size n, the number of cells that are non-blank on M's tape is bounded by $p(n)$. Problem that belong to PSPACE are usually called PSPACE-*easy*. On the other hand, some problems are known to be *complete* for the class PSPACE, and are regarded as the most difficult problems in PSPACE. Formally, a problem P is PSPACE-*hard*, iff any problem in PSPACE can be reduced to P in polynomial time. In practice, to prove that a problem P is PSPACE hard, it is sufficient to exhibit a polynomial-time reduction from a known PSPACE-hard problem to P (this is what we are about to do in this section). Then, a problem is PSPACE-*complete* iff it is PSPACE-hard and in PSPACE. Remark that the class PSPACE contains the class NP, and it

is widely believed that NP $\subsetneq$ PSPACE, and that all PSPACE-complete problems are outside NP [24].

In this section, we will reduce our scheduling problem MSS from the *(language) universality problem for (finite state labeled) automata* to establish the PSPACE-hardness of MSS. The universality problem is well-known to be PSPACE-complete [23]. As we are about to discuss the complexity of MSS, we need to fix an encoding for the input to this problem. Each instance of MSS is defined by a (finite) system of sporadic tasks τ *and* a scheduler Run. Sporadic tasks are described by natural numbers, so we use the standard Boolean encoding for them. A naive encoding for the scheduler would consist in describing it as the set of all pairs $(S, \mathsf{Run}(S))$, where S is a state of $A(\tau, \mathsf{Run})$. Intuitively, this would mean that we represent Run as a table that returns, for each reachable state S of the system, the set of tasks to schedule. However, in this case, the *size* of (the encoding of) the scheduler would already be as large as the Baker-Cirinei automaton $A(\tau, \mathsf{Run})$, and MSS could be solved in *linear time* (w.r.t. to the size of the scheduler). Clearly, such a definition of the scheduler is not realistic in practice, and does not capture the hardness of the problem. This computational hardness of MSS comes from the fact that, in general, the size of $A(\tau, \mathsf{Run})$ is *exponential* in the size of (the encoding of) τ, as previously remarked by Baker and Cirinei [3]. So, we need to adopt an encoding of the scheduler s.t. the size of $A(\tau, \mathsf{Run})$ is *exponential* in the size of Run too. For that purpose, we assume in the sequel that schedulers are given as *Turing machines whose size is polynomial in the size of the system of tasks* τ. Intuitively, this means that the scheduler has to be given as an *algorithm* that actually *computes* the set of tasks to be scheduled and does not merely look up a table spanning all the reachable states to take a decision.

Let us now start our discussion of the complexity of MSS by formally defining the *universality problem*.

3.1 The universality problem for labeled automata

We begin by extending the notion of automaton given in Definition 1. For that purpose, we need to extend automata with a *labeling function*, that associates, to each automaton state, a symbol from a finite alphabet Σ. Following the classical definitions in language theory, we call a *word* (on Σ) a finite (and potentially empty) sequence $a_1 a_2 \cdots a_n$ of symbols from Σ. We denote by Σ^+ the set of all non-empty *words* on Σ. Then:

Definition 15 A *labeled automaton* is a tuple $A = \langle V, E, S_0, F, \Sigma, \lambda \rangle$, s.t. $\langle V, E, V_0, F \rangle$ is an automaton (see Definition 1), Σ is a finite *alphabet*, and $\lambda : V \mapsto \Sigma$ is a *labeling function of the states*.

Throughout this section, we implicitly assume that all the automata we consider are *labeled* and *finite*. For an automaton A, we denote by $|A|$ the size of A, i.e. the size $|V|$ of its set of states. The definitions of *path* and *reachable states* carry on to the case of labeled automata. We also let, for all set of states $V' \subseteq V$ and all letter $a \in \Sigma$, $\mathsf{post}_a(V')$ be the set of a-labeled successors of V', i.e. $\mathsf{post}_a(V') \stackrel{\mathbf{def}}{=} \{v \mid \lambda(v) = a \text{ and } \exists v' \in V' : (v', v) \in E\}$. For all path $\rho = v_1, v_2, \ldots, v_\ell$, we abuse notations and denote by $\lambda(\rho)$ the *label* of ρ, i.e. the finite word $\lambda(v_1)\lambda(v_2)\cdots\lambda(v_n) \in \Sigma^+$. In a labelled automaton $A = \langle V, E, V_0, F, \Sigma, \lambda \rangle$, a path $\rho = v_1, v_2, \ldots, v_\ell$ is *accepting* iff $v_1 \in V_0$ and $v_\ell \in F$. A word $w \in \Sigma^+$ is *accepted* by A iff there exists an accepting path ρ_w of A s.t. $\lambda(\rho_w) = w$. Remark that there could be several paths labeled by

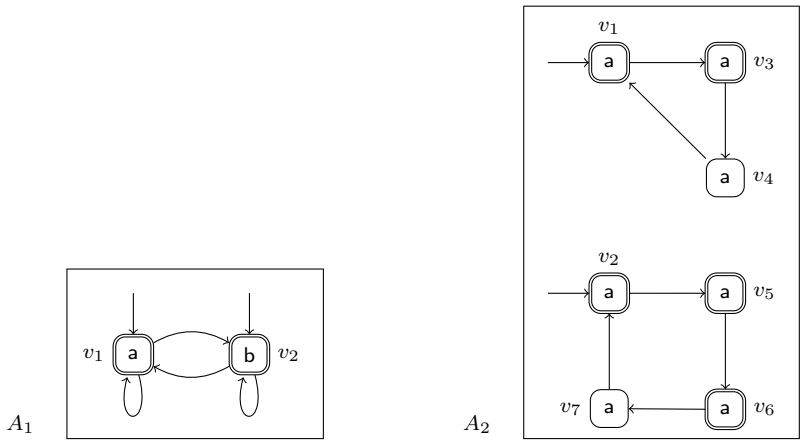


Fig. 4 Two labeled (and complete) automata A_1 and A_2. A_1 is deterministic and universal. A_2 is not deterministic and not universal.

w, that are accepting or not. We only require, following the classical definitions of non-deterministic finite automata, that *there exists* an accepting path on a given word to accept it. Finally, the *language* of a labeled automaton A is the set $L(A) \subseteq \Sigma^+$ of all the words accepted by A. Without loss of generality, we assume that all labeled automata $A = \langle V, E, V_0, F, \Sigma, \lambda \rangle$ we consider in this section are *complete*, i.e. that for all $a \in \Sigma$: (i) there is $v \in V_0$ s.t. $\lambda(v) = a$ and (ii) for all $v \in V$, there exists $v' \in V$ s.t. $(v, v') \in E$ and $\lambda(v') = a$. That is, there is an initial state labeled by each $a \in \Sigma$, and each state admits an a-successor for all $a \in \Sigma$. Remark that it is always possible, to *complete* a labeled automaton while preserving its accepted language: for $A = \langle V, E, V_0, F, \Sigma, \lambda \rangle$, let A' be the labeled automaton $\langle V', E', V_0', F, \Sigma, \lambda' \rangle$ where $V' = V \uplus V_\Sigma$, $V_\Sigma = \{v_a \mid a \in \Sigma\}$, $E' = E \cup (V' \times V_\Sigma)$, $V_0' = V_0 \cup V_\Sigma$ and λ is s.t. $\lambda'(v) = \lambda(v)$ for all $v \in V$ and $\lambda'(v_a) = a$ for all $v_a \in \Sigma$. It is easy to check that A' is complete. Moreover, $L(A') = L(A)$ as we have added only non-accepting states, and edges towards those non-accepting states (hence we have added no accepting path), and the size of A' is linear in the size of A.

Example 3 Fig. 4 displays two examples of labeled and complete automata, called A_1 (on alphabet $\Sigma_1 = \{\mathsf{a}, \mathsf{b}\}$) and A_2 (on alphabet $\Sigma_2 = \{\mathsf{a}\}$). States v are depicted as circles, bearing their label $\lambda(v)$. The names of the states are indicated next to the states. States with double circles are accepting. Initial states bear an input arrow with no state as source. This graphical convention will be used throughout the paper. An example of run of A_1 is $v_1 v_2 v_1 v_1$. It accepts abaa. It is easy to see $L(A_1) = \Sigma_1^+$, i.e. A_1 accepts all non-empty words on Σ_1. An example of run of A_2 is $v_2 v_5 v_6$, which accepts aaa. Another run of A_2 is $v_2 v_5 v_6 v_7$, which *rejects* aaaa. Nevertheless, the word aaaa is not rejected by A_2, as *there exists* an accepting run of A_2 on that word: $v_1 v_3 v_4 v_1$. Still, some words are not accepted by A_2. It is easy to check that A_2 accepts all words from Σ_2^+, *except* the words whose length is $12 \times k$ for some $k \geqslant 1$. Indeed, starting from v_1, the automaton can accept any word in Σ_1^+ but the words whose length is a multiple of 3. Similarly, from v_2, all the words whose length is not a multiple of 4 can be accepted. Hence the rejected words are exactly those whose length is both a multiple of 3 and 4, hence the words whose length is a multiple of 12.

The problem we will be concerned with, in this section, is the *language universality problem*[5]:

Problem 2 (Universality) Given a labeled automaton A, does $L(A) = \Sigma^+$?

When the answer is positive, we say that A *is universal*. It is well known that this problem is PSPACE-complete [23].

Example 4 Automaton A_1 in Fig. 4 is universal, while automaton A_2 is not (see the discussion in Example 3).

Solving the universality problem via the complement. Let us briefly discuss some classical results about determinising and complementing finite automata to solve the universality problem. These results are present in many basic textbooks about language theory but we recall them for the sake of completeness and, most of all, to fix notations.

A labeled automaton $A = \langle V, E, V_0, F, \Sigma, \lambda \rangle$ is *deterministic* iff: (i) for all $a \in \Sigma$, there exists one and only one state $v \in V_0$ s.t. $\lambda(v) = a$, and (ii) for all state v, for all $a \in \Sigma$, there exists one and only one $v' \in V$ s.t. $\lambda(v') = a$ and $(v, v') \in E$. This implies that A is *complete* and that, for each word $w \in \Sigma^+$ there is exactly one path in A labeled by w (be it accepting or not). It is well-known that automata can be *determinised*, i.e. that, for each labeled automaton A one can build a deterministic automaton A' s.t. $L(A) = L(A')$. More precisely, for all (complete) automaton A, let $\mathsf{Det}(A)$ be the automaton $\langle \mathcal{V}, \mathcal{E}, \mathcal{V}_0, \mathcal{F}, \Sigma, \lambda' \rangle$ where:

1. $\mathcal{V} = \{S \subseteq V \mid S \neq \emptyset \wedge \forall v_1, v_2 \in S : \lambda(v_1) = \lambda(v_2)\}$;
2. $(S_1, S_2) \in \mathcal{E}$ iff there is $a \in \Sigma$ s.t. $S_2 = \mathsf{post}_a(S_1)$;
3. $S \in \mathcal{V}_0$ iff there exists $a \in \Sigma$ s.t. $S = \{v \in V_0 \mid \lambda(v) = a\}$;
4. $\mathcal{F} = \{S \in \mathcal{V} \mid S \cap F \neq \emptyset\}$;
5. for all $S \in \mathcal{V}$: $\lambda'(S) = a$ iff for all $v \in S$: $\lambda(v) = a$.

Remark, that, by construction, the number of states of $\mathsf{Det}(A)$ is, in the worst case, exponential in the number of states of A. Remark further that, since we have assumed A to be complete, there will be no $S \subseteq V$ and no $a \in \Sigma$ s.t. $\mathsf{post}_a(S) = \emptyset$. Hence, the restriction that $\emptyset$ is not a state of $\mathsf{Det}(A)$ in point 1 is sound.

A nice property of *deterministic automata* is that they can be easily *complemented*. Formally, for all labeled automata $A = \langle V, E, S_0, F, \Sigma, \lambda \rangle$, let $\overline{A}$ be the complement of A, defined as the automaton $\langle V, E, S_0, V \setminus F, \Sigma, \lambda \rangle$, i.e. the automaton obtained from A by swapping accepting and non-accepting states. In the case where A is deterministic, it is easy to see that $\overline{A}$ is deterministic too (as the complement does not change the structure of the automaton), and accepts the complement of A's language, i.e. $\Sigma^+ \setminus L(A)$. Remark that this is not true in general for (possibly non-deterministic) labeled automata. The following proposition summarises these results:

Proposition 3 *For all labeled automata A, $\mathsf{Det}(A)$ is a deterministic labeled automaton s.t. $L(\mathsf{Det}(A)) = L(A)$ and $|\mathsf{Det}(A)| \leqslant 2^{|A|}$. For all* deterministic *labeled automata A, $\overline{A}$ is a deterministic labeled automaton s.t. $L(\overline{A}) = \Sigma^+ \setminus L(A)$ and $|\overline{A}| = |A|$. Thus, for all labeled automaton A, $\overline{\mathsf{Det}(A)}$ is a* deterministic *automaton s.t. $L(\overline{\mathsf{Det}(A)}) = \Sigma^+ \setminus L(A)$ and $\left|\overline{\mathsf{Det}(A)}\right| \leqslant 2^{|A|}$.*

[5] Remark that the classical definition of the universality problem asks whether the automaton recognises Σ^*. Our definition problem differs slightly, as in our setting, automata bear labels on the nodes, instead of the edges. Hence, no such labeled automaton can accept ε. However, it is straightforward to check that this does not change the complexity of the problem.

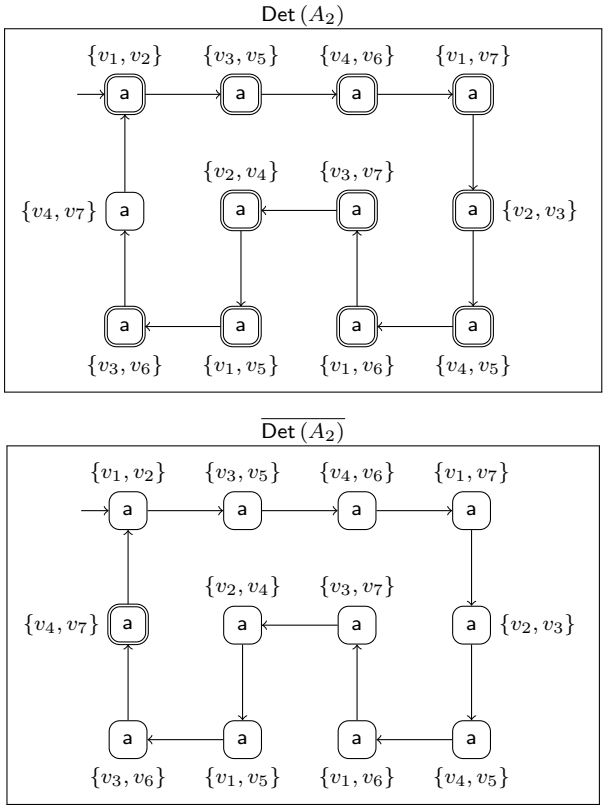


Fig. 5 The automata $\mathsf{Det}\,(A_2)$ and $\overline{\mathsf{Det}\,(A_2)}$.

Example 5 Fig. 5 presents the automata $\mathsf{Det}\,(A_2)$ and $\overline{\mathsf{Det}\,(A_2)}$. It is easy to check that both are deterministic, that $L(\mathsf{Det}\,(A_2)) = L(A_2)$ and that $L(\overline{\mathsf{Det}\,(A_2)}) = \Sigma^+ \setminus L(A_2)$, i.e. $\overline{\mathsf{Det}\,(A_2)}$ accepts exactly the words that A_2 rejects.

As a direct consequence, we obtain a procedure for solving *language universality*, as a labeled automaton A is universal iff $L(\overline{\mathsf{Det}\,(A)}) = \emptyset$. Thus, determining whether an automaton A is universal amounts to *finding an accepting path* in $\overline{\mathsf{Det}\,(A)}$. In general, in order to find such an accepting path in an automaton B, we can restrict ourselves to *loopless paths*. Indeed, if $\rho = v_0, v_1, \ldots, v_n$ is an accepting path s.t. there are two positions i and j with $0 \leqslant i < j \leqslant n$ and $v_i = v_j$, then the path $\rho' = v_0, \ldots, v_i, v_{j+1}, \ldots, v_n$ is an accepting path in B too. Repeating this construction leads to a loopless accepting path, i.e. a path $\rho^* = v_0, \ldots, v_\ell$ s.t. for all i, j: $0 \leqslant i < j \leqslant n$ implies that $v_i \neq v_j$. Clearly, such a path is of length at most $|B|$. In the case where $B = \overline{\mathsf{Det}\,(A)}$ for some A, we know, by Proposition 3, that $|B|$ is at most $2^{|A|}$. Hence:

Corollary 1 *For all labeled automata A: A is universal iff there is no accepting path of length at most $2^{|A|}$ in $\overline{\mathsf{Det}\,(A)}$.*

Example 6 The automaton $\overline{\mathsf{Det}\,(A_2)}$ (where A_2 is the automaton given in Fig. 4) is obtained by swapping accepting and non-accepting states in $\mathsf{Det}\,(A_2)$, shown in Fig. 5.

Clearly, in $\overline{\mathsf{Det}(A_2)}$, any run of length $12 \times k$ (for $k \geqslant 1$) is an accepting run. However, a single run of length 12 is sufficient to capture the fact that $\{v_4, v_7\}$ is reachable in $\overline{\mathsf{Det}(A_2)}$. We conclude that $\overline{\mathsf{Det}(A_2)} \neq \emptyset$. Thus, $\mathsf{Det}(A_2)$ is not universal, nor is A_2.

This means that we have reduced the universality problem for labeled automata to finding a run (of bounded length) reaching a certain set of states (the accepting states of $\overline{\mathsf{Det}(A)}$) in an automaton. This question is thus similar to MSS, which amounts to finding a run reaching Fail_τ in the Baker-Cirinei automaton $A(\tau, \mathsf{Run})$. This observation will be the basis of our reduction to prove PSPACE-hardness of MSS (see Section 3.2).

3.2 From the universality problem to MSS

In this section, we show how to build, from any automaton $B = \langle V, E, S_0, F, \Sigma, \lambda \rangle$, a task system τ_B, and a scheduler f_B s.t. τ_B *is schedulable under* f_B *iff* B *is universal*. As stated above, the intuition of the reduction relies on Corollary 1, i.e., we can solve the universality problem for labeled automata by reducing it to a reachability problem in an automaton, where we only need to consider runs of bounded length. Thus, our encoding of the universality problem on B will consists in building τ_B and f_B is such a way that (i) the executions of τ_B under f_B *simulate* all the paths of $\overline{\mathsf{Det}(B)}$ that are of length at most $2^{|B|}$; and (ii) that the accepting states of $\overline{\mathsf{Det}(B)}$ correspond to the states of the real-time system where a deadline is missed. Thus, roughly speaking, $A(\tau_B, f_B)$ will *miss a deadline iff an accepting state is reachable by a path of length at most* $2^{|B|}$ *in* $\overline{\mathsf{Det}(B)}$. By Corollary 1, the latter means that B is not universal, which sets the reduction.

Actually, an *exact simulation* of the runs of $\overline{\mathsf{Det}(B)}$ by the real-time system will not be possible. That is, all runs of $\overline{\mathsf{Det}(B)}$ will be simulated by an execution of τ_B under f_B, but some executions of this real-time system will not correspond to genuine runs of $\overline{\mathsf{Det}(B)}$. This is due to the fact that we will use the tasks to encode the states of $\overline{\mathsf{Det}(B)}$, and that requests of the (sporadic) tasks are non-deterministic. However, it will be the duty of the scheduler to check that the execution of τ_B corresponds to a genuine run of $\overline{\mathsf{Det}(B)}$ and to *reset* the system when it is not the case (the exact meaning of the reset will be made clear later). Hence, for each execution of the real-time system, we have three possibilities. Either, it corresponds to a run of $\overline{\mathsf{Det}(B)}$ and this run is *accepting*. In this case, the system will miss a deadline. Or, it corresponds to a run of $\overline{\mathsf{Det}(B)}$ which is *not accepting*. In this case, the system will never miss a deadline. Or does not correspond to a run of $\overline{\mathsf{Det}(B)}$. In that case, the scheduler will ensure that the system never misses a deadline, in order to avoid *false positive*, i.e. to avoid that the real-time system misses and deadline because it has reached a failure state by execution that does not correspond to a run of $\overline{\mathsf{Det}(B)}$. Let us now give formally the reduction.

The system of sporadic tasks τ_B. Let us fix labeled automaton $B = \langle V, E, S_0, F, \Sigma, \lambda \rangle$. Let us assume that $V = \{v_1, v_2, \ldots, v_{|B|}\}$, and that $\overline{\mathsf{Det}(B)} = \langle \mathcal{V}, \mathcal{E}, \mathcal{V}_0, \mathcal{F}, \Sigma, \Lambda \rangle$. Then, let $\tau_B = \{\tau_1, \tau_2, \ldots, \tau_{|B|}, \tau_{|B|+1}\}$ be the set of sporadic tasks s.t.:

1. for all $1 \leqslant i \leqslant |B|$: $C_i = 1$ and $T_i = 3$.
2. for all $1 \leqslant i \leqslant |B| + 1$: $D_i = T_i$.
3. $C_{|B|+1} = 1$ and $T_{|B|+1} = 2^{|B|} + 4$.

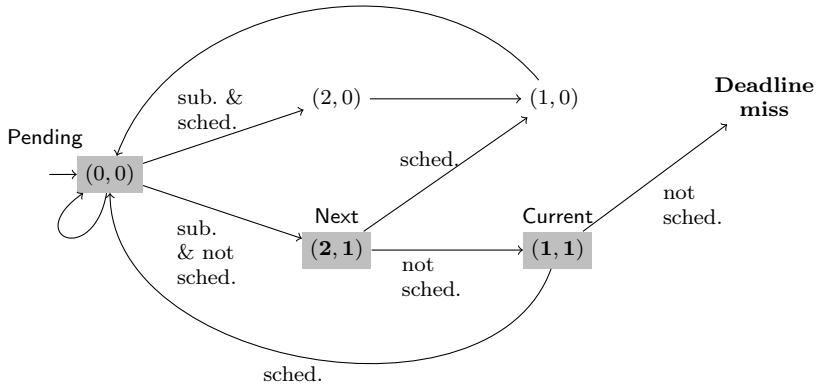


Fig. 6 Depiction of the life cycle of a task τ_i that corresponds to an automaton state (i.e. $1 \leqslant i \leqslant |A|$). States are pairs $(\mathrm{nat}, \mathrm{rct})$. sched. means « the task gets scheduled on a CPU » and sub. means « the task submits a new job ». Bold font has been used for the states where the task is active. Next, Current and Pending refer to the sets defined hereunder.

Thus, the system has exactly one task τ_i per automaton state v_i, plus an extra task $\tau_{|B|+1}$ that we will use to encode the fact that an execution of the real-time system corresponds to a genuine run of the automaton. Observe that each task τ_i with $1 \leqslant i \leqslant |B|$ can be in only a few different states. The life cycle of such a task is depicted in Fig. 6.

We rely on those different possible states for each task in order to encode information about the run of $\overline{\mathsf{Det}\,(B)}$ that is simulated by the execution of the real-time system. Namely, each state S of $A(\tau_B, f_B)$ encodes *two states* of $\overline{\mathsf{Det}\,(B)}$, that we denote $\mathsf{Current}(S)$ and $\mathsf{Next}(S)$. Intuitively, $\mathsf{Current}(S)$ is the current state of $\overline{\mathsf{Det}\,(B)}$ and $\mathsf{Next}(S)$ is a potential successor of $\mathsf{Current}(S)$ in $\overline{\mathsf{Det}\,(B)}$ (as said earlier, it will be the duty of the scheduler to check whether this successor is genuine or not). Moreover, $\mathsf{Pending}(S)$ contains all the states corresponding to a task that can submit a job. Formally:

- $\mathsf{Current}(S) \stackrel{\mathbf{def}}{=} \{v_i \in V \mid \mathrm{rct}_S(\tau_i) = \mathrm{nat}_S(\tau_i) = 1\}$,
- $\mathsf{Next}(S) \stackrel{\mathbf{def}}{=} \{v_i \in V \mid \mathrm{rct}_S(\tau_i) = 1 \wedge \mathrm{nat}_S(\tau_i) = 2\}$ and
- $\mathsf{Pending}(S) \stackrel{\mathbf{def}}{=} \{v_i \in V \mid \mathrm{rct}_S(\tau_i) = \mathrm{nat}_S(\tau_i) = 0\}$.

We also rely on the following definitions to characterise the relationship between states of the real-time system and states of the deterministic automaton:

- $\mathsf{Start}(S)$ holds if and only if $\mathrm{rct}_S(\tau_{|B|+1}) = 1$ and $\mathrm{nat}_S(\tau_{|B|+1}) = T_{|B|+1}$. That is, we consider that the real-time system starts to simulate the automaton when task $\tau_{|B|+1}$ submits a job. In this case, the scheduler will have to check that $\mathsf{Next}(S)$ is an initial state of $\overline{\mathsf{Det}\,(B)}$.
- $\mathsf{Valid}(S)$ holds if and only if $\mathrm{rct}_S(\tau_{|B|+1}) = 1$. Along a run of the real-time systems, states will be *valid* as long as the corresponding run in $\overline{\mathsf{Det}\,(B)}$ is genuine.
- $\mathsf{End}(S)$ holds if and only if $\mathrm{rct}_S(\tau_{|B|+1}) = 1$ and $\mathrm{nat}_S(\tau_{|B|+1}) = 1$. This allow us to detect states where $\tau_{|B|+1}$ has to be scheduled to avoid missing its deadline.

– $\mathsf{CmpRejects}(S)$ holds if and only if $\mathsf{Current}(S) \notin \mathcal{F}$. Thus, $\mathsf{CmpRejects}(S)$ holds iff the state of $\overline{\mathsf{Det}\,(B)}$ that is encoded by S is *not accepting*. When B is *universal*, we expect to visit only states s.t. $\mathsf{CmpRejects}(S)$ *does not hold* along all valid simulations of a run of B.

The scheduler f_B. Let us now define the scheduler associated to B. As said earlier, the submission of jobs by the tasks can be regarded as a non-deterministic *guess* of a possible successor of the current state of the automaton. The duty of the scheduler is then to *check* whether this guess is correct or not, and to ensure that a deadline is *missed* iff an *accepting* state of $\overline{\mathsf{Det}\,(B)}$ is reached during the *correct* simulation of a run of $\overline{\mathsf{Det}\,(B)}$. This means that the scheduler must ensure that no deadline is ever missed whenever the simulation is *not correct*, in order to avoid *false positives*.

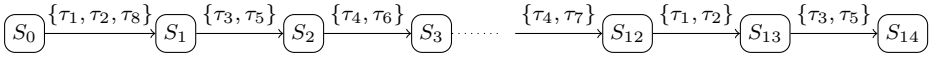
This intuition is formalised by the following rules (which are commented hereunder). Let S be a state of the real-time system. Then, $f_B(S)$ is the scheduler on $|B|+1$ CPUs, computed as follows:

1. If $\mathsf{Valid}(S) \wedge \neg\mathsf{Start}(S) \wedge \neg\mathsf{End}(S) \wedge \mathsf{CmpRejects}(S)$, then:
   (a) If either $\mathsf{Current}(S) = \emptyset$ and $\mathsf{Next}(S) \in \mathcal{V}_0$, or $(\mathsf{Current}(S), \mathsf{Next}(S)) \in \mathcal{E}$, then $f_B(S) = \{\tau_i \mid v_i \in \mathsf{Current}(S)\}$.
   (b) Otherwise, $\mathsf{Run}(S) = \mathsf{Active}(S)$.
2. If $\mathsf{Start}(S)$, then $f_B(S) = \{\tau_i \mid v_i \in \mathsf{Current}(S) \cup \mathsf{Next}(S)\}$
3. If $\mathsf{End}(S)$, then $f_B(S) = \mathsf{Active}(S)$.
4. If $\neg\mathsf{Valid}(S)$, then $f_B(S) = \mathsf{Active}(S)$.
5. If $\mathsf{Valid}(S) \wedge \neg\mathsf{Start}(S) \wedge \neg\mathsf{End}(S) \wedge \neg\mathsf{CmpRejects}(S)$, then $f_B(S) = \emptyset$.

As can be seen from these rules, in each state S s.t. $\mathsf{Valid}(S)$ holds, the scheduler checks whether, either (i) $\mathsf{Current}(S) = \emptyset$ and $\mathsf{Next}(S)$ is an initial state of $\overline{\mathsf{Det}\,(B)}$, or that $\mathsf{Next}(S)$ is a successor of $\mathsf{Current}(S)$ in $\overline{\mathsf{Det}\,(B)}$ (rule 1a). Intuitively, this means that the *guess is correct* in this state. If it is the case, the scheduler allocates the CPU to all the tasks τ_j that encode $\mathsf{Current}(S)$. In that case, it is easy to check that, in each successor state S' of S: $\mathsf{Current}(S') = \mathsf{Next}(S)$ (see Fig. 6). Moreover when moving from S to S', some tasks that where *pending* in S have submitted new jobs, which encode in S' a new potential successor $\mathsf{Next}(S')$. Thus, $\mathsf{Next}(S') \subseteq \mathsf{Pending}(S)$. Finally, all the tasks that encoded a *current* state in S are now pending in S', i.e. $\mathsf{Pending}(S') = \mathsf{Pending}(S) \setminus \mathsf{Next}(S') \cup \mathsf{Current}(S)$. Thus, along a valid simulation, the scheduler always ensures that the tasks are in either of the three highlighted states in Fig. 6. On the other hand, if the *guess is incorrect* (rule 1b), the scheduler schedules all the active tasks, including $\tau_{|B|+1}$. This moves the system to successor state S' s.t. $\mathsf{Valid}(S')$ is *false*.

In states where $\mathsf{Valid}(S)$ is false, the scheduler always allocates the CPUs to all the active tasks (rule 4) in order to ensure that no deadline is missed. This is to avoid *false positive*, i.e., deadline misses that occur although the system has not simulated a genuine run of $\overline{\mathsf{Det}\,(B)}$.

Additionally, when the scheduler witnesses a state S in which the set of tasks τ' has submitted a job, and where $\tau_{|B|+1} \in \tau'$ (that is, $\mathsf{Start}(S)$ holds), it will give the CPU to all the tasks that encode either a *current* or a *next* state (rule 2). This is to ensure that, in all successor state S' of S, all the tasks in τ' (except $\tau_{|B|+1}$) will encode a state in $\mathsf{Next}(S')$, and that $\mathsf{Current}(S') = \emptyset$. From there, the scheduler will check that $\mathsf{Next}(S')$ indeed encodes an initial state of the automaton, and the simulation will start as described above (rule 1a). Next, if the state S satisfies $\mathsf{End}(S)$, then, we have



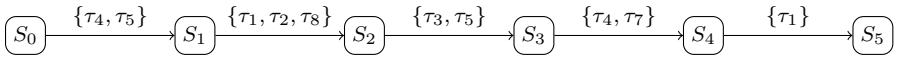
	S_0	S_1	S_2	S_3	$\cdots$	S_{12}	S_{13}	S_{14}
τ_1	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$	$\langle 0,0\rangle$		$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$
τ_2	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$	$\langle 0,0\rangle$		$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$
τ_3	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$		$\langle 1,1\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$
τ_4	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\cdots$	$\langle 2,1\rangle$	$\langle 1,1\rangle$	$\langle \mathbf{0},\mathbf{1}\rangle$
τ_5	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$		$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$
τ_6	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$		$\langle 1,1\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$
τ_7	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$		$\langle 2,1\rangle$	$\langle 1,1\rangle$	$\langle \mathbf{0},\mathbf{1}\rangle$
τ_8	$\langle 0,0\rangle$	$\langle 131,1\rangle$	$\langle 130,1\rangle$	$\langle 129,1\rangle$		$\langle 120,1\rangle$	$\langle 119,1\rangle$	$\langle 118,1\rangle$
Next	$\emptyset$	$\{v_1, v_2\}$	$\{v_3, v_5\}$	$\{v_4, v_6\}$		$\{v_4, v_7\}$	$\{v_1, v_2\}$	$\{v_3, v_5\}$
Current	$\emptyset$	$\emptyset$	$\{v_1, v_2\}$	$\{v_3, v_5\}$		$\{v_3, v_6\}$	$\{v_4, v_7\}$	$\{v_1, v_2\}$
Valid	F	T	T	T		T	T	T
End	F	F	F	F	$\cdots$	F	F	F
CmpRejects	T	T	T	T		T	$\mathbf{F}$	T
$\in \mathsf{Fail}_{\tau_{A_2}}$?	*no*	*no*	*no*	*no*		*no*	*no*	**yes**

Fig. 7 An example of a run of $A(\tau_{A_2}, f_{A_2})$ that simulates a run of A_2.

simulated so far a run of length at least $2^{|B|}$ and $\tau_{|B|+1}$ is about to miss its deadline. In this case, the scheduler schedules all the tasks to move to an *invalid* state (rule 3), even if the successor guess was correct. However, this is not a problem as far as the simulation of B is concerned, as we know that, if a state v of $\overline{\mathsf{Det}\,(B)}$ is reachable, then, there exists a run of length at most $2^{|B|}$ that allows to reach it (see Corollary 1). It is thus not useful to extend the simulation any further, as all the reachable states of $\overline{\mathsf{Det}\,(B)}$ will be captured by runs of length at most $2^{|B|}$. Finally, when a *valid* state S is reached s.t. $\mathsf{Current}(S)$ encodes an accepting state of $\overline{\mathsf{Det}\,(B)}$, B is *not universal* and the scheduler prevents all the tasks from getting access to the CPU (rule 5), to ensure that a deadline will be missed.

Example 7 Fig. 7 shows an example of a run $S_0, S_1, \ldots, S_{13}, S_{14}$ of $A(\tau_{A_2}, f_{A_2})$ that simulates the (unique) run of length 13 of $\overline{\mathsf{Det}\,(A_2)}$ (see Fig. 4 and Fig. 5). The diagram on the top of the figure depicts the run. The set of tasks that label the edges are the set of tasks that submit a job when moving from one state to another (remark that the intermediate states are not shown). The top part of the figure gives, for each task τ_i, and each state S_j, the pair of values of $\langle \mathrm{nat}_{S_j}(\tau_i), \mathrm{rct}_{S_j}(\tau_i)\rangle$. The bottom part of the table shows, for each state S_j, the values of $\mathsf{Next}(S_j)$, $\mathsf{Current}(S_j)$, and so forth. The last line of that table indicates whether each state S_j is a failure state. It is easy to see that, although we have defined τ_{A_2} and f_{A_2} from A_2, this run actually simulates a run of $\overline{\mathsf{Det}\,(A_2)}$. When moving from S_0 to S_1, an intermediate state S_0' s.t. $\mathsf{Start}(S_0')$ holds is met and the scheduler applies rule 2. When moving from S_i to S_{i+1} for all $1 \leqslant i \leqslant 11$, the scheduler applies rule 1a. Remark that the accepting state $\{v_4, v_7\}$ of $\overline{\mathsf{Det}\,(A_2)}$ is indeed *detected* in S_{13}, as $\mathsf{CmpRejects}(S_{13})$ is *false*. In this state (and since $\mathsf{Valid}(S_{13})$ holds and $\mathsf{End}(S_{13})$ does not hold), the scheduler applies rule 5, and does not schedule any task. Then, all the tasks in $\mathsf{Current}(S_{13})$ (i.e., τ_4 and τ_7) miss their deadline in state S_{14}.

Example 8 In Example 7 we have shown a run of $A(\tau_{A_2}, f_{A_2})$ that faithfully simulates a run of A_2. Let us now show an example of run that fails to do so, because, at some



	S_0	S_1	S_2	S_3	S_4	S_5
τ_1	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$	$\langle 0,0\rangle$	$\langle 2,0\rangle$
τ_2	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$
τ_3	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$	$\langle 0,0\rangle$
τ_4	$\langle 0,0\rangle$	$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,0\rangle$
τ_5	$\langle 0,0\rangle$	$\langle 2,0\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$
τ_6	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,1\rangle$	$\langle 0,0\rangle$
τ_7	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 2,1\rangle$	$\langle 1,0\rangle$
τ_8	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 131,1\rangle$	$\langle 130,1\rangle$	$\langle 129,1\rangle$	$\langle 128,0\rangle$
Next	$\emptyset$	$\emptyset$	$\{v_1, v_2\}$	$\{v_3, v_5\}$	$\{v_4, v_7\}$	$\emptyset$
Current	$\emptyset$	$\emptyset$	$\emptyset$	$\{v_1, v_2\}$	$\{v_3, v_5\}$	$\emptyset$
Valid	F	F	T	T	T	**F**
End	F	F	F	F	F	F
CmpRejects	T	T	T	T	T	T
$\in \mathsf{Fail}_{\tau_{A_2}}$?	*no*	*no*	*no*	*no*	*no*	*no*

Fig. 8 An example of a run of $A(\tau_{A_2}, f_{A_2})$ that fails to simulate a run of A_2.

point, the *successor guess is incorrect*. The run is depicted in Fig. 8, with the same conventions as in Example 7. When moving from S_0 to S_1, the system traverses an intermediate state S' (not shown in the figure) in which $\mathrm{nat}_{S'}(\tau_8) \neq T_8$, as τ_8 has not submitted a job. Hence, the simulation has not started (i.e. $\mathsf{Start}(S')$ is false), S' is an *invalid* state, the scheduler applies rule 4, and schedules all the active tasks (here τ_4 and τ_5). In the resulting state S_1, no task is active, and no state of $\overline{\mathsf{Det}(A_2)}$ is encoded by S_1. Moreover $\mathsf{Valid}(S_1)$ does not hold. However, when moving from S_1 to S_2, τ_8 submits a job. Hence, the intermediate state S'' between S_1 and S_2 (not shown in the figure) is valid and s.t. $\mathsf{Start}(S'')$ holds. Thus, the scheduler applies rule 2. The resulting state S_2 is then *valid*, and the guess made in S_2 is *correct*, as $\mathsf{Current}(S_2) = \emptyset$ and $\mathsf{Next}(S_2) = \{v_1, v_2\}$ is an initial state of $\overline{\mathsf{Det}(A_2)}$. The scheduler proceeds with rule 1a and the run visits S_3 where again the guess is correct, as $\{v_3, v_5\}$ is indeed a successor of $\{v_1, v_2\}$. Thus, rule 1a is applied again to reach S_4. In this state, however, $\mathsf{Next}(S_4) = \{v_4, v_7\}$ is *not a valid successor* of $\{v_3, v_5\}$ (see Fig. 5). Thus, the guess made there is *not correct*. This is detected by the scheduler that applies rule 1b, and schedules all the active tasks (including τ_8). As a consequence $\mathrm{rct}_{S_5}(\tau_8) = 0$ and S_5 is an *invalid* state, i.e. $\mathsf{Valid}(S_5)$ is false. From that state, the scheduler will always give the CPU to all the active tasks, unless τ_8 submits a new job, and we start a fresh simulation. This guarantees that no deadline will be missed, and allows to avoid *false positive*.

Let us now better characterise the cases where a deadline will be missed in the system of tasks τ_B, running under f_B:

Lemma 1 *For all labeled automaton B, Fail_{τ_B} is reachable in $A(\tau_B, f_B)$ iff there exists a state S s.t. $\mathsf{Valid}(S)$, $\neg\mathsf{End}(S)$ and $\neg\mathsf{CmpRejects}(S)$ that is reachable in $A(\tau_B, f_B)$.*

Proof First, observe that, for all $S \in \mathsf{Reach}(A(\tau_B, f_B))$, $\mathsf{Start}(S)$ does not hold. Indeed, if $\mathsf{Start}(S)$ holds then $\mathrm{nat}_S(\tau_{|B|+1}) = T_{|B|+1}$. However, no task can have its nat equal to its period in a reachable state of $A(\tau_B, f_B)$. Indeed, assume $\tau_{|B|+1}$ submits a job in a

reachable state S (hence $\mathrm{nat}_S(\tau_{|B|+1}) \leqslant 0$). Then, all successors S' of S are obtained by letting one clock tick elapse after the task submission, hence $\mathrm{nat}_{S'}(\tau_{|B|+1}) = T_{|B|+1} - 1$ in all such S'.

Now, assume there is a reachable state S s.t. $\mathsf{Valid}(S) \wedge \neg\mathsf{End}(S) \wedge \neg\mathsf{CmpRejects}(S)$. Since $\mathsf{CmpRejects}(S)$ does not hold, we conclude that $\mathsf{Current}(S) \neq \emptyset$. Indeed, since $\emptyset \notin \mathcal{F}$ by definition of $\overline{\mathsf{Det}\,(B)}$, $\mathsf{Current}(S) = \emptyset$ would imply that $\mathsf{CmpRejects}(S)$ holds. On the other hand, since $\mathsf{Valid}(S)$ holds, we know that $\mathrm{rct}_S(\tau_{|B|+1}) = 1$, hence $\mathrm{nat}_S(\tau_{|B|+1}) \geqslant 1$. Thus, $\tau_{|B|+1}$ cannot submit a job in S. Let τ' be a set of tasks that are eligible in S, and let S' be the state s.t. $S \xrightarrow{\tau'} S'$ (remark that S' is not part of the reachable states of $A(\tau_B, f_B)$). As $\tau_{|B|+1} \notin \tau'$, $\mathrm{nat}_{S'}(\tau_{|B|+1}) \neq T_{|B|+1}$, thus $\mathsf{Start}(S')$ does not hold. Remark that $\mathsf{Current}(S') = \mathsf{Current}(S)$ and that $\mathsf{Next}(S') = \mathsf{Next}(S)$. Then, the scheduler applies rule 5, and does not schedule any task. Hence, all tasks in $\mathsf{Current}(S')$ have a negative laxity in all successors S'' of S', since $\mathrm{rct}_{S'}(\tau_i) = \mathrm{nat}_{S'}(\tau_i) = 1$ for all τ_i s.t. $v_i \in \mathsf{Current}(S')$, by definition of $\mathsf{Current}(S') = \mathsf{Current}(S)$. Remark that such a task always exists as $\mathsf{Current}(S') = \mathsf{Current}(S) \neq \emptyset$. Thus S'' is reachable and in Fail_{τ_B}.

We finish the proof by showing the reverse direction, *per absurdum*. Let S be a state in Fail_{τ_B} that is reachable in $A(\tau_B, f_B)$, by a path $S^0 S^1 \dots S^\ell$ (that is $S^\ell = S$). W.l.o.g., assume that for all $0 \leqslant i \leqslant \ell - 1$: $S^i \notin \mathsf{Fail}_{\tau_B}$, that is, S^ℓ is the first state to miss a deadline along the path. Let us show, by contradiction, that $\mathsf{Valid}(S^{\ell-1}) \wedge \neg\mathsf{End}(S^{\ell-1}) \wedge \neg\mathsf{CmpRejects}(S^{\ell-1})$. Assume it is not the case, thus, either $\neg\mathsf{Valid}(S^{\ell-1})$ or $\mathsf{End}(S^{\ell-1})$ or $\mathsf{CmpRejects}(S^{\ell-1})$. We consider all those cases separately, by letting $T^{\ell-1}$ and τ' be respectively the (intermediate) state and the set of tasks s.t. $S_{\ell-1} \xrightarrow{\tau'} T^{\ell-1} \xrightarrow{f_B} S^\ell$:

- If $\neg\mathsf{Valid}(S^{\ell-1})$, we consider two sub-cases.
  - If $\neg\mathsf{Valid}(T^{\ell-1})$, then the scheduler applies rule 4 and schedules all the active tasks in $T^{\ell-1}$. Thus, if there exists τ s.t. $\mathrm{laxity}_{S^\ell}(\tau) < 0$, then $\mathrm{laxity}_{S^{\ell-1}}(\tau) < 0$ too and $S^{\ell-1} \in \mathsf{Fail}_{\tau_B}$. Contradiction.
  - If $\mathsf{Valid}(T^{\ell-1})$, then $\tau_{|B|+1}$ has just submitted a job, i.e. $\tau_{|B|+1} \in \tau'$. In this case, $\mathsf{Start}(T^{\ell-1})$ holds. Now, let τ_i be a task s.t. $\mathrm{laxity}_{S^\ell}(\tau_i) < 0$. Such a task must exist since $S^\ell \in \mathsf{Fail}_{\tau_B}$ by hypothesis. Moreover, $\mathrm{rct}_{S^\ell}(\tau_i) > 1$, and $\mathrm{laxity}_{S^{\ell-1}}(\tau_i) \geqslant 0$, as we have assumed that $S^{\ell-1} \notin \mathsf{Fail}_{\tau_B}$. We consider all the possible cases for τ_i and, in each case, derive a contradiction. First, assume $\tau_i \in \tau'$. Since, by definition $T_i > C_i$, $\mathrm{laxity}_{S^\ell}(\tau_i) > 0$. Contradiction. Second, assume τ_i encodes a state in $\mathsf{Next}(S^{\ell-1}) = \mathsf{Next}(T^{\ell-1})$. In that case, $\mathrm{rct}_{S^{\ell-1}}(\tau_i) = \mathrm{rct}_{T^{\ell-1}}(\tau_i) < \mathrm{nat}_{S^{\ell-1}}(\tau_i) = \mathrm{nat}_{T^{\ell-1}}(\tau_i)$. Thus, $\mathrm{laxity}_{S^\ell}(\tau_i) > 0$. Contradiction. Finally, let us assume that τ_i encodes a state in $\mathsf{Current}(S^{\ell-1}) = \mathsf{Current}(T^{\ell-1})$. Since $\mathsf{Start}(T^{\ell-1})$ holds, the scheduler applies rule 2, and schedules all the tasks encoding a state in $\mathsf{Current}(T^{\ell-1})$. As $\mathrm{rct}_{T^{\ell-1}}(\tau_i) = 1$, we conclude that $\mathrm{rct}_{S^\ell}(\tau_i) = 0$, hence $\mathrm{laxity}_{S^\ell}(\tau_i) \geqslant 0$. Contradiction.
- If $\mathsf{End}(S^{\ell-1})$, then, $\mathsf{End}(T^{\ell-1})$ holds too, and the scheduler schedules all active tasks, by rule 3. Hence, if there exists τ s.t. $\mathrm{laxity}_{S^\ell}(\tau) < 0$, then $\mathrm{laxity}_{S^{\ell-1}}(\tau) < 0$ too and $S^{\ell-1} \in \mathsf{Fail}_{\tau_B}$. Contradiction.
- Finally if $\mathsf{CmpRejects}(S^{\ell-1})$, we also assume, following the previous points of the proof, that $\mathsf{Valid}(S^{\ell-1}) \wedge \neg\mathsf{End}(S^{\ell-1})$. We consider two cases.
  - If *either* $\mathsf{Current}(S) = \emptyset$ and $\mathsf{Next}(S) \in \mathcal{V}_0$, *or* $(\mathsf{Current}(S), \mathsf{Next}(S)) \in \mathcal{E}$, the scheduler schedules all the tasks that encode a state in $\mathsf{Current}(S^{\ell-1}) =$

$\mathsf{Current}(T^{\ell-1})$, by rule 1a. By similar reasoning as above, $\tau_{|B|+1}$ cannot account for a deadline miss in S^ℓ since $\neg\mathsf{End}(S^{\ell-1})$, all tasks τ_i encoding a state in $\mathsf{Next}(S)$ are s.t. $\mathrm{nat}_{T^{\ell-1}}(\tau_i) > \mathrm{rct}_{T^{\ell-1}}(\tau_i)$ and thus cannot account for a deadline miss in S^ℓ either, and all tasks encoding a state in $\mathsf{Current}(S^{\ell-1}) = \mathsf{Current}(T^{\ell-1})$ get scheduled. We conclude that all tasks have a positive laxity in S^ℓ, hence $S^\ell \notin \mathsf{Fail}_{\tau_B}$. Contradiction.

– If *neither* $\mathsf{Current}(S) = \emptyset$ and $\mathsf{Next}(S) \in \mathcal{V}_0$ *nor* $(\mathsf{Current}(S), \mathsf{Next}(S)) \in \mathcal{E}$, the scheduler schedules all the active tasks, by rule 1b, hence no task can have a negative laxity in S^ℓ, unless $S^{\ell-1} \in \mathsf{Fail}_{\tau_B}$. Contradiction. □

Correctness of the construction. The intuition that we have just sketched suggests that our reduction is *sound*: all runs of $A(\tau_B, f_B)$ that traverse *valid states* should *faithfully* simulate a run of $\overline{\mathsf{Det}\,(B)}$. More precisely, if a state S with $\mathsf{Valid}(S) = \mathit{true}$ is reachable in $A(\tau_B, f_B)$, then we have the guarantee that $\mathsf{Current}(S)$ is actually a reachable state of $\overline{\mathsf{Det}\,(B)}$. This is formalised by the next Lemma:

Lemma 2 *For all labeled automaton B, for all state $S \in \mathsf{Reach}(A(\tau_B, f_B))$: $\mathsf{Valid}(S)$ and $\mathsf{Current}(S) \neq \emptyset$ implies that $\mathsf{Current}(S) \in \mathsf{Reach}(\overline{\mathsf{Det}\,(B)})$.*

Proof Recall that we assume that $\overline{\mathsf{Det}\,(B)} = \langle \mathcal{V}, \mathcal{E}, \mathcal{V}_0, \mathcal{F}, \Sigma, \Lambda\rangle$. Let $S_0, S_1, \ldots, S_\ell$ be a run of $A(\tau_B, f_B)$ s.t. $\mathsf{Valid}(S_\ell)$. Let k be the least position s.t. $k \leqslant \ell$, and for all $k \leqslant j \leqslant \ell$: $\mathsf{Valid}(S_j)$ holds. That is, $S_k, S_{k+1}, \ldots, S_\ell$ is the longest suffix of $S_0, S_1, \ldots, S_\ell$ that visits only valid states. Remark that $k \geqslant 1$, as $\mathsf{Valid}(S_0)$ does not hold. Since $\mathsf{Valid}(S_k)$ holds but $\mathsf{Valid}(S_{k-1})$ does not, task $\tau_{|B|+1}$ has made a request on the transition from S_{k-1} to S_k. Thus, by definition of f_B (rule 2), $\mathsf{Current}(S_k) = \emptyset$. Moreover, since $\mathsf{Valid}(S_k)$ and $\mathsf{Valid}(S_{k+1})$ hold, $\mathsf{Next}(S_k) = \mathsf{Current}(S_{K+1}) \in \mathcal{V}_0$, by rule 1a. Then, we consider the sequence of states $V_0, V_1, \ldots V_{\ell-k-1}$ s.t. for all $0 \leqslant i \leqslant \ell - k - 1$: $V_i = \mathsf{Current}(S_{k+i+1})$. Since $\mathsf{Valid}(S_i)$ holds for all $k \leqslant i \leqslant \ell$, and since $\mathsf{Current}(S_{k+1}) \in \mathcal{V}_0$, $V_0, V_1, \ldots V_{\ell-k-1}$ is a run of $\overline{\mathsf{Det}\,(B)}$. Thus, in particular, $V_{\ell-k-1} = \mathsf{Current}(S_\ell)$ is in $\mathsf{Reach}(\overline{\mathsf{Det}\,(B)})$. □

In general, not all runs of $\overline{\mathsf{Det}\,(B)}$ can be simulated by $A(\tau_B, f_B)$. This is because, in a run $V_0 V_1 \ldots V_\ell$ of $\overline{\mathsf{Det}\,(B)}$, there could be $0 \leqslant i < \ell$ s.t. $V_i \cap V_{i+1} \neq \emptyset$. Unfortunately, with our encoding, $\mathsf{Current}(S) \cap \mathsf{Next}(S) = \mathsf{Current}(S) \cap \mathsf{Pending}(S) = \mathsf{Next}(S) \cap \mathsf{Pending}(S) = \emptyset$ for all states S of $A(\tau_B, f_b)$. Intuitively, this is shown in Fig. 6: when a task encodes a Current state, it has to go trough the Next and Pending states (highlighted on the figure) before reaching Current again. However, this difficulty can be overcome by an easy technical construction that we present now. Roughly speaking, it consists of augmenting the set of states of the automaton by a value $i \in \{0, 1, 2\}$, whose value will be incremented (modulo 3) along each path in the automaton. Let $B = \langle V, E, S_0, F, \Sigma, \lambda\rangle$ be a labeled automaton. We let $\mathsf{Triple}\,(B)$ be the automaton $\langle V \times \{0,1,2\}, E', S_0 \times \{0\}, F \times \{0,1\}, \Sigma, \lambda'\rangle$, where $((v_1, i_1), (v_2, i_2)) \in E'$ iff $(v_1, v_2) \in E$ and $i_2 = \big((i_1 + 1) \mod 3\big)$; and for all $v \in V$, $i \in \{0,1,2\}$: $\lambda((v,i)) = \lambda(v)$. Finally, for all state $v = (q, i) \in V$, we denote by $\mathsf{col}\,(v)$ the value i of the extra information, and call this value the *color* of v. The interest of this construction is given by the following Lemma, which states that (i) the construction preserves the accepted language; (ii) it increases the size of automaton only by a factor 3; and (iii) it guarantees that, along all runs $V_0, V_1, \ldots, V_n$ of $\mathsf{Det}\,(\mathsf{Triple}\,(B))$, whenever a state v of $\mathsf{Triple}\,(B)$ appears in some state V_i (for $i \leqslant n-2$), it does not appear in the

two next states V_{i+1}, V_{i+2}, which is the property we need to guarantee that $A(\tau_B, f_B)$ can simulate all runs of the automaton:

Lemma 3 *For all automata $B = \langle V, E, S_0, F, \Sigma, \lambda \rangle$:*

1. $L(\mathsf{Triple}(B)) = L(A)$;
2. $|\mathsf{Triple}(B)| = 3 \times |B|$;
3. *For all path $V_0, V_1, \ldots, V_n$ of $\mathsf{Det}(\mathsf{Triple}(B))$, for all $0 \leqslant i \leqslant n-2$: $V_i \cap V_{i+1} = V_{i+1} \cap V_{i+2} = V_i \cap V_{i+2} = \emptyset$.*

Proof Points 1 and 2 are direct by construction. In order to establish point 3, we prove, by induction on i, a stronger statement. We prove first that for all $0 \leqslant i \leqslant n$: for all $v_1, v_2 \in V_i$: $\mathsf{col}(v_1) = \mathsf{col}(v_2)$. Moreover, letting $\mathsf{col}(V_i) = k$ iff $\mathsf{col}(v) = k$ for all $v \in V$, we prove that $\mathsf{col}(V_i) = (\mathsf{col}(V_{i+1}) + 1) \mod 3$. Intuitively, this means that all the $\mathsf{Triple}(B)$ states that are present in a state V_i, along a run of $\mathsf{Det}(\mathsf{Triple}(B))$, all share the same color, which we denote by $\mathsf{col}(V_i)$, and that, along the run of $\mathsf{Det}(\mathsf{Triple}(B))$, this color cycles among the values 0, 1, and 2. This is proved by induction on the length of the run.

For the base case, we let $i = 0$. By definition of $\mathsf{Triple}(B)$ and $\mathsf{Det}(\mathsf{Triple}(B))$: for all $v \in V_0$: $\mathsf{col}(v) = 0$. Hence, $\mathsf{col}(V_0) = 0$.

For the inductive case, we let $i = \ell > 1$, and we assume that there is $k \in \{0, 1, 2\}$ s.t. for all $s \in v_{\ell-1}$: $\mathsf{col}(v) = k$, hence $\mathsf{col}(V_{\ell-1}) = k$. By definition of $\mathsf{Det}(\mathsf{Triple}(B))$, $V_\ell = \mathsf{post}_a(V_{\ell-1})$ for some a. Thus, all the states v in V_ℓ are the successor of some state v' in V_{ℓ_1}. However all states $v' \in V_{\ell-1}$ are s.t. $\mathsf{col}(v') = k$, by induction hypothesis. Hence, by construction of $\mathsf{Triple}(B)$, for all $v \in V_\ell$: $\mathsf{col}(c) = (k+1) \mod 3$.

Let us now conclude the proof and establish point 3. Let us consider V_i and $V_i + 1$ for some $0 \leqslant i < n$ and let us show that $V_i \cap V_{i+1} = \emptyset$. By the property proved above, $\mathsf{col}(V_i) \neq \mathsf{col}(V_{i+1})$, which implies that $V_i \cap V_{i+1} = \emptyset$. The cases $V_i \cap V_{i+2} = \emptyset$ and $V_{i+1} \cap V_{i+2} = \emptyset$ are proved similarly. □

Then, let us prove this construction is sufficient to ensure that our reduction is *complete*, i.e. that, when applying the construction of the set of sporadic tasks, and of the scheduler, to the automaton $\mathsf{Triple}(B)$ (instead of B), yields a Baker-Cirinei automaton $A(\tau_{\mathsf{Triple}(B)}, f_{\mathsf{Triple}(B)})$ that can simulate all the runs of $\overline{\mathsf{Det}(\mathsf{Triple}(B))}$:

Lemma 4 *For all labeled automaton B, for all $V \in \mathsf{Reach}(\overline{\mathsf{Det}(\mathsf{Triple}(B))})$, there is $S \in \mathsf{Reach}(A(\tau_{\mathsf{Triple}(B)}, f_{\mathsf{Triple}(B)}))$ s.t. $\mathsf{Current}(S) = V$, $\mathsf{Valid}(S)$ and $\neg\mathsf{End}(S)$.*

Proof Throughout this proof, and in order to alleviate the notations, we denote the automaton $\mathsf{Triple}(B)$ by C and assume that $\overline{\mathsf{Det}(C)} = \left\langle \mathcal{V}^C, \mathcal{E}^C, \mathcal{V}_0^C, \mathcal{F}^C, \Sigma, \Lambda^C \right\rangle$. The proof consists in building, for each run of $\overline{\mathsf{Det}(C)}$, a corresponding run of $A(\tau_C, f_C)$ that reaches a state S with the desired properties. Let us consider a run $V_0, V_1, \ldots V_\ell$ of $\overline{\mathsf{Det}(C)}$. Since $V_0, V_1, \ldots V_\ell$ is a run, $(V_i, V_{i+1}) \in \mathcal{E}^C$ for all $0 \leqslant i < \ell$. By Corollary 1 and Lemma 3, we can assume that $\ell \leqslant 2^{|C|}$, as those runs are sufficient to capture all reachable states in $\overline{\mathsf{Det}(C)}$. From $V_0, V_1, \ldots V_\ell$, we build a run $S_0, S_1, S_2, \ldots, S_{\ell+2}$ of $A(\tau_C, f_C)$ s.t., for all $0 \leqslant i < \ell + 2$, $S_i \xrightarrow{\tau^i} S'_i \xrightarrow{f_B} S_{i+1}$ where:

- S_0 is the initial state of $A(\tau_C, f_C)$ and
- $\tau^0 = \{\tau_j \mid v_j \in V_0\} \cup \{\tau_{|C|+1}\}$ and
- for all $1 \leqslant i < \ell + 1$: $\tau^i = \{\tau_j \mid v_j \in V_i\}$ and

– $\tau^{\ell+1} \subseteq \mathsf{Eligible}(S_{\ell+1})$.

First remark that, since $\tau_{|C|+1} \in \tau^0$, $\mathsf{Start}(S'_0)$ holds. Thus, the scheduler applies rule 2, and $\mathrm{nat}_{S_1}(\tau_{|C|+1}) = 2^{|C|} + 4 - 1 = 2^{|C|} + 3$. Hence, for all $1 \leqslant i \leqslant \ell + 2$: $\mathrm{nat}_{S_i}(\tau_{|C|+1}) = 2^{|C|} + 4 - i > 1$ since $\ell \leqslant 2^{|C|}$. We conclude that, for all $1 \leqslant i \leqslant \ell+2$: $\mathsf{End}(S_i)$ does not hold.

Let us first show, by induction on $i = 1, 2, \ldots, \ell+1$, that the prefix $S_0, S_1, \ldots, S_{\ell+1}$ is a genuine run of $A(\tau_C, f_C)$ s.t. for all $1 \leqslant i \leqslant \ell + 1$: $\mathsf{Valid}(S_i)$ holds, $\mathsf{Next}(S_i) = V_{i-1}$, $\mathsf{Current}(S_i) = V_{i-2}$ (assuming V_{-1} denotes $\emptyset$), $\mathsf{Current}(S_i) = \emptyset$ iff $i = 1$ and $V = \mathsf{Pending}(S_i) \cup \mathsf{Next}(S_i) \cup \mathsf{Current}(S_i)$.

Base case $i = 1$**.** By definition of τ^0, and since $\mathsf{Start}(S'_0)$ holds, the scheduler applies rule 2. Hence $\mathsf{Next}(S_1) = \{v_i \mid \tau_i \in \tau^0 \setminus \{\tau_{|C|+1}\}\} = V_0$, $\mathsf{Current}(S_1) = \emptyset$, and, as all the tasks are idle in S_0, $V = \mathsf{Pending}(S_1) \cup \mathsf{Current}(S_1) \cup \mathsf{Next}(S_1)$. Moreover, since $\tau_{|C|+1} \in \tau^0$: $\mathrm{rct}_{\tau_{|C|+1}} = 1$ and $\mathsf{Valid}(S_1)$ holds.

Inductive case $i = k > 1$**.** By induction hyp.: $\mathsf{Valid}(S_{k-1})$ holds, $\mathsf{Next}(S_{k-1}) = V_{k-2}$, $\mathsf{Current}(S_{k-1}) = V_{k-3}$ (where, again $V_{-1} = \emptyset$), $\mathsf{Current}(S_{k-1}) = \emptyset$ iff $k = 2$ and $V = \mathsf{Pending}(S_{k-1}) \cup \mathsf{Current}(S_{k-1}) \cup \mathsf{Next}(S_{k-1})$. We first check that all the tasks in τ^{k-1} can submit a job in S_{k-1}. For that purpose, we prove that $\tau^{k-1} \subseteq \{\tau_i \mid v_i \in \mathsf{Pending}(S_{k-1})\}$, i.e., all tasks in τ^{k-1} encode a *pending state* (see Fig. 6). This is sufficient, as, by definition, all those tasks have their nat and rct equal to 0 and can thus submit a job. Since $V = \mathsf{Pending}(S_{k-1}) \cup \mathsf{Current}(S_{k-1}) \cup \mathsf{Next}(S_{k-1})$ by induction hypothesis, and since $\mathsf{Pending}(S_{k-1})$, $\mathsf{Current}(S_{k-1})$ and $\mathsf{Next}(S_{k-1})$ are disjoint sets by definition, $\mathsf{Pending}(S_{k-1}) = V \setminus (\mathsf{Current}(S_{k-1}) \cup \mathsf{Next}(S_{k-1}))$, i.e., a task encoding a state is *pending* iff it does not encode a state from Next of $\mathsf{Current}$. Thus, $\tau^{k-1} \subseteq \{\tau_i \mid v_i \in \mathsf{Pending}(S_{k-1})\}$ holds if $\tau^{k-1} \cap \{\tau_i \mid v_i \in \mathsf{Next}(S_{k-1}) \cup \mathsf{Current}(S_{k-1})\}$, i.e. all the tasks that submit are not encoding a *next* or *current state*. Since τ^{k-1} encodes the set V_{k-1} by definition, since $\mathsf{Next} = V_{k-2}$ and since $\mathsf{Current} = V_{k-3}$, by induction hypothesis, this amounts to check that $V_{k-1} \cap (V_{k-2} \cup V_{k-3}) = \emptyset$, which is true by Lemma 3.

Next, since $\mathsf{Valid}(S_{k-1})$ holds and $\mathsf{End}(S_{k-1})$ does not, then $\mathrm{rct}_{S_{k-1}}(\tau_{|C|+1}) = 1$ and $\mathrm{nat}_{S_{k-1}}(\tau_{|C|+1}) > 1$. Let us now establish that S_k has all the desired properties. We consider two cases. On the one hand, if $\mathsf{Current}(S_{k-1}) = \emptyset$ then $k = 2$ by induction hypothesis. Since $\mathsf{Next}(S_{k-1}) = V_{k-2}$ by induction hypothesis, we conclude that $\mathsf{Next}(S_{k-1}) = V_0$. On the other hand, if $\mathsf{Current}(S_{k-1}) \neq \emptyset$ then $k > 2$, and $\big(\mathsf{Current}(S_{k-1}), \mathsf{Next}(S_{k-1})\big) = (V_{k-3}, V_{k-2})$ by induction hypothesis. However, $(V_{k-3}, V_{k-2}) \in \mathcal{E}^C$, as $V_0, V_1, \ldots, V_\ell$ is a run of $\overline{\mathsf{Det}\,(C)}$. In both cases, we conclude that the scheduler applies rule 1a in S'_{k-1}. Hence, all the tasks encoding a state in $\mathsf{Current}(S_{k-1}) = \mathsf{Current}(S'_{k-1})$ are scheduled and their rct and nat becomes zero, i.e. $\mathsf{Current}(S_{k-1}) \subseteq \mathsf{Pending}(S_k)$. All the tasks encoding a state in $\mathsf{Next}(S_{k-1})$ are not scheduled and become current, i.e. $\mathsf{Current}(S_k) = \mathsf{Next}(S_{k-1}) = V_{k-2} \neq \emptyset$. All the tasks that have submitted in S_{k-1} are not scheduled either and encode a *next* state in S_k, i.e. $\mathsf{Next}(S_k) = \{v_i \mid \tau_i \in \tau^{k-1}\} = V_{k-1}$. Clearly, all tasks in τ^{k-1} that have submitted in S_{k-1} encode a *pending* state in S_{k-1}, i.e. $\tau^{k-1} \subseteq \{\tau_i \mid v_i \in \mathsf{Pending}(S_{k-1})\}$. This holds because, by induction hypothesis, $V = \mathsf{Pending}(S_{k-1}) \cup \mathsf{Next}(S_{k-1}) \cup \mathsf{Current}(S_{k-1})$, and the tasks that encode a state in $\mathsf{Next}(S_{k-1}) \cup \mathsf{Current}(S_{k-1})$ cannot

submit a job, by definition of those sets. Thus:

$$
\begin{aligned}
&\mathsf{Pending}(S_k)\\
&\quad= \mathsf{Pending}(S_{k-1}) \setminus \{v_i \mid \tau_i \in \tau^{k-1}\} \cup \mathsf{Current}(S_{k-1}) && \text{See above}\\
&\quad= V \setminus \mathsf{Current}(S_{k-1}) \setminus \mathsf{Next}(S_{k-1}) \setminus \{v_i \mid \tau_i \in \tau^{k-1}\} \cup \mathsf{Current}(S_{k-1}) && \text{By H.I}\\
&\quad= V \setminus \mathsf{Current}(S_{k-1}) \setminus \mathsf{Next}(S_{k-1}) \setminus \mathsf{Next}(S_k) \cup \mathsf{Current}(S_{k-1}) && \text{Def. of } \tau^{k-1}\\
&\quad= V \setminus \mathsf{Current}(S_{k-1}) \setminus \mathsf{Current}(S_k) \setminus \mathsf{Next}(S_k) \cup \mathsf{Current}(S_{k-1}) && \text{See above}\\
&\quad= V \setminus \mathsf{Current}(S_{k-1}) \setminus \big(\mathsf{Current}(S_k) \cup \mathsf{Next}(S_k)\big) \cup \mathsf{Current}(S_{k-1})\\
&\quad= V \setminus \big(\mathsf{Current}(S_k) \cup \mathsf{Next}(S_k)\big)
\end{aligned}
$$

The last equality above holds[6], because $\mathsf{Current}(S_{k-1}) \cap \big(\mathsf{Current}(S_k) \cup \mathsf{Next}(S_k)\big) = V_{k-3} \cap (V_{k-2} \cup V_{k-1}) = \emptyset$, by Lemma 3. Hence, $V = \mathsf{Pending}(S_k) \cup \mathsf{Current}(S_k) \cup \mathsf{Next}(S_k)$, as these three sets are disjoint by definition. Finally, $\tau_{|C|+1}$ is not scheduled, thus $\mathrm{rct}_{S_k}(\tau_{|C|+1}) = \mathrm{rct}_{S_{k-1}}(\tau_{|C|+1}) = 1$. Hence, $\mathsf{Valid}(S_k)$ holds. This concludes the induction.

Thus, $\mathsf{Next}(S_{\ell+1}) = V_\ell$, $\mathsf{Current}(S_{\ell+1}) = V_{\ell-1}$ and $\mathsf{Valid}(S_{\ell+1})$. Hence, in $S'_{\ell+1}$, the scheduler applies rule 1a, and schedules only those tasks that are encoding a state in $\mathsf{Current}(S_{\ell+1})$. We conclude that $\mathsf{Current}(S_{\ell+2}) = \mathsf{Next}(S_{\ell+1}) = V_\ell$. Moreover, since $\mathsf{Valid}(S_{\ell+1})$ holds and $\mathsf{End}(S_{\ell+1})$ does not, we conclude that $\mathsf{Valid}(S_{\ell+2})$ holds too. Hence the Lemma. □

Thanks to those lemmata we can now prove that the construction of $\tau_{\mathsf{Triple}(B)}$ and $f_{\mathsf{Triple}(B)}$ from B is an effective reduction of the universality problem for labeled automata to the MSS problem:

Proposition 4 *For all labeled automata $B = \langle V, E, V_0, F, \Sigma, \lambda\rangle$: B is universal iff $\mathsf{Fail}_{\tau_{\mathsf{Triple}(B)}}$ is not reachable in $A(\tau_{\mathsf{Triple}(B)}, f_{\mathsf{Triple}(B)})$.*

Proof To alleviate notations, we let $C = \mathsf{Triple}\,(B)$ and assume that $\overline{\mathsf{Det}\,(\mathsf{Triple}\,(B))} = \left\langle \mathcal{V}^C, \mathcal{E}^C, \mathcal{V}_0^C, \mathcal{F}^C, \Sigma, \Lambda^C \right\rangle$. We prove the equivalent statement that B *is not* universal iff Fail_{τ_C} *is* reachable in $A(\tau_C, f_C)$. We establish the two directions of the iff separately.

Assume B is not universal. Then, there is $w \in \Sigma^+$ s.t. $w \notin L(B)$. Hence, $w \in L(\overline{\mathsf{Det}\,(C)})$, by Lemma 3, and by definition of the complement and the determinisation of a labeled automaton. Thus, there is a state $V \in \mathsf{Reach}(\overline{\mathsf{Det}\,(C)})$ that is accepting in $\overline{\mathsf{Det}\,(C)}$, i.e. $V \in \mathcal{F}^C$. Let V^* be such a state. By Lemma 4, there is $S^* \in \mathsf{Reach}(A(\tau_C, f_C))$ s.t. $\mathsf{Current}(S^*) = V^*$, and $\mathsf{Valid}(S^*)$, and $\neg\mathsf{End}(S^*)$ hold (remark that Lemma 4 holds for all labeled automaton, thus in particular for C). Moreover, since $\mathsf{Current}(S^*) = V^* \in \mathcal{F}^C$, $\mathsf{CmpRejects}(S^*)$ does not hold. Thus, S^* is a reachable state of $A(\tau_C, f_C)$ s.t. $\mathsf{Valid}(S^*)$, $\neg\mathsf{End}(S^*)$ and $\neg\mathsf{CmpRejects}(S^*)$. Hence, by Lemma 1, Fail_{τ_C} is reachable in $A(\tau_C, f_C)$.

For the reverse direction, assume that Fail_{τ_C} *is* reachable in $A(\tau_C, f_C)$. Let S^* be a reachable state of $A(\tau_C, f_C)$ s.t. $\mathsf{Valid}(S^*) \wedge \neg\mathsf{End}(S^*) \wedge \neg\neg\mathsf{CmpRejects}(S^*)$. By Lemma 1, such a state exists, as we have assumed that Fail_{τ_C} is reachable in $A(\tau_C, f_C)$. Remark that, since $\neg\mathsf{CmpRejects}(S^*)$ holds, $\mathsf{Current}(S^*) \in \mathcal{F}^C$. Hence, $\mathsf{Current}(S^*) \neq \emptyset$, by definition of the determinisation of labeled automata. Thus, by Lemma 2, $\mathsf{Current}(S^*) \in \mathsf{Reach}(\overline{\mathsf{Det}\,(C)})$. Hence, $\mathsf{Current}(S^*)$ is an accepting and reachable state of $\overline{\mathsf{Det}\,(C)}$, hence $L(\overline{\mathsf{Det}\,(C)}) \neq \emptyset$, and $L(\mathsf{Det}\,(C))$ is not universal. Since $L(\mathsf{Det}\,(C)) = L(B)$ by Lemma 3, and definition of the determinisation of labeled automata, we conclude that B is not universal either. □

[6] Remark that, in general $V \setminus A \setminus B \cup A = V \setminus B$ iff $A \cap B = \emptyset$.

PSPACE *completeness.* We have now at our disposal all the necessary results to prove that MSS is PSPACE-complete.

Theorem 1 MSS *is* PSPACE*-complete*

Proof First, we show the membership to PSPACE. For that purpose, we briefly describe a Turing machine that recognises pairs (τ, Run), where τ is a system of sporadic tasks, and Run is a scheduler, iff τ is *not* schedulable under Run (that is, a machine that recognises the complement of MSS). The machine guesses non-deterministically a path of $A(\tau, \mathsf{Run})$ and checks whether this path reaches a state from Fail_τ. More precisely, the machine starts by writing S_0 on its tape, where S_0 is the initial state of $A(\tau, \mathsf{Run})$. Then, it enters a loop that consists in (i) guessing a state S' of $A(\tau, \mathsf{Run})$ and writing S' on the tape next to the current state S, (ii) checking whether S' is a genuine successor of S in $A(\tau, \mathsf{Run})$ and (iii) erasing S from the tape. The machine accepts iff the current state is in Fail_τ. It is easy to check that this machine needs at most a space on its tape that is polynomial in the size of τ, to carry out this computation. Hence, the complement of MSS is in NPSPACE. However, by Savitch's theorem PSPACE =NPSPACE, so the complement of MSS is in PSPACE [24]. Thus, there exists a *deterministic* polynomial-space Turing machine M that recognises the complement of MSS. As this machine is *deterministic*, its complement $\overline{M}$ is also a polynomial-space Turing machine, and it recognises the complement of the language of M, which is MSS. Hence MSS is in PSPACE.

Proposition 4 proves that the construction of $\tau_{\mathsf{Triple}(B)}$ and $f_{\mathsf{Triple}(B)}$ from B is an effective reduction of the universality problem for labeled automata to MSS. It is easy to see that this reduction can be carried out in polynomial time. That is, the rules that define $f_{\mathsf{Triple}(B)}$ can easily be encoded in a Turing machine whose size is polynomial in the size of B. In particular, remark that the constant $T_{|B|+1}$ requires only a polynomial number of bits, when using the standard binary encoding. As the universality problem is PSPACE-complete [23], MSS is PSPACE-hard.

Thus, MSS is both PSPACE-hard and PSPACE-easy. It is thus PSPACE-complete. □

Discussion. Let us briefly discuss this complexity result. First, we note that, to the best of our knowledge, this is the first precise characterisation of the complexity of MSS. This result, however, could be refined, as it holds for a very general case of scheduler. Indeed, to perform the reduction, we have relied on the scheduler f_B, which is not one of the standards schedulers of the literature (such as global EDF, or global DM). Moreover, it is important to note that f_B is not *work-conserving*. Indeed, since there are as many processors as tasks, τ_B, as it is defined, is always feasible, but f_B sometimes need to let tasks miss their deadlines, in particular when it has detected an accepting run of $\overline{\mathsf{Det}\,(\mathsf{Triple}\,(B))}$, i.e. B is not universal. So, it could be the case that MSS has a lower complexity when only certain special kinds of scheduler are considered. We leave these questions open for future works.

As recalled in the introduction of this section, it is known that NP $\subseteq$ PSPACE [24]. It is also widely believed that NP $\subsetneq$ PSPACE, and that, in particular, all PSPACE-complete problems are outside NP[7]. Thus, under these hypothesis, our result implies (i) that an algorithm using an amount of memory that is *polynomial* in the size of the

[7] In some sense, the status of 'NP versus PSPACE' is similar to that of 'P versus NP', as it is known that P $\subseteq$ NP, and it is believed that P $\subsetneq$ NP and that all NP-complete problems are not in P.

system's description is sufficient to solve MSS[8], and (*ii*) that even testing that a given activation sequence of the tasks leads to a deadline miss is not feasible in polynomial time (because this would imply that MSS is in NP too, and thus that NP = PSPACE). Intuitively, MSS is thus much harder a problem than, for instance, the Hamiltonian path problem, or the Boolean satisfiability problem, that are both NP-complete [14, 24]. Also, and still under the hypothesis that NP $\subsetneq$ PSPACE, our result rules out the possibility of finding a 'easily computable analytical test[9]' that would be a sufficient *and* necessary test for MSS.

Although this result might sound rather negative, we believe it is nevertheless interesting, as it provides insights on the structure of the problem, and can guide us when looking for heuristics to solve this problem efficiently in practice. In the field of *computer-aided verification*, and, in particular, *automata-based model-checking*, several PSPACE-complete problems (such as the universality problem we have discussed above) have been well-studied, as they are most relevant in practice. In this setting, a recent line of research [11, 13, 9] has proposed to use *antichain-based* techniques to solve, efficiently, several problems that belong to this class. In the rest of the paper, we apply those techniques to the MSS problem, and show that they allow to dramatically reduce the portion of the Baker-Cirinei automaton that needs to be explored, in practice.

4 Solving the reachability problem

Let us now discuss techniques to solve the reachability problem. Let $A = \langle V, E, S_0, F\rangle$ be an automaton. For any $S \in V$, let $\mathsf{Succ}(S) = \{S' \mid (S, S') \in E\}$ be the set of one-step successors of S. For a set of states R, we let $\mathsf{Succ}(R) = \cup_{S\in R}\mathsf{Succ}(S)$. Then, solving the reachability problem on A can be done by a *breadth-first traversal* of the automaton, as shown in Algorithm 1.

Algorithm 1: Breadth-first traversal.

```
1 begin
2     i ← 0 ;
3     R_0 ← S_0 ;
4     repeat
5         i ← i + 1 ;
6         R_i ← R_{i−1} ∪ Succ(R_{i−1}) ;
7         if R_i ∩ F ≠ ∅ then return Reachable ;
8     until R_i = R_{i−1};
9     return Not reachable ;
```

Intuitively, for all $i \geqslant 0$, R_i is the set of states that are reachable from S_0 in i steps at most. The algorithm computes the sets R_i up to the point where (*i*) either a state from F is met or (*ii*) the sequence of R_i stabilises because no new states have been discovered, and we declare F to be unreachable. This algorithm always terminates and returns the correct answer. Indeed, either F is reachable in, say k

[8] Remark that this was already known by the result of Baker and Cirinei [3].

[9] Such as the well-known $U \leqslant 1$ for the mono-processor version of EDF, which is computable in polynomial time.

steps, and then $R_k \cap F \neq \emptyset$, and we return '`Reachable`'. Or F is not reachable, and the sequence eventually stabilises because $R_0 \subseteq R_1 \subseteq R_2 \subseteq \cdots \subseteq V$, and V is a finite set. Then, we exit the loop and return '`Not reachable`'. Remark that this algorithm has the advantage that the whole automaton does not need to be stored in memory before starting the computation, as Definition 11 and Definition 12 allow us to compute $\mathsf{Succ}(S)$ *on the fly* for any state S. Nevertheless, in the worst case, this procedure needs to explore the whole automaton and is thus in $\mathcal{O}(|V|)$ which can be too large to handle in practice [3].

Equipped with such a simple definition of *automaton*, this is the best algorithm we can hope for. However, in many practical cases, the set of states of the automaton is endowed with a *strong semantic* that can be exploited to speed up Algorithm 1. In our case, states are tuples of integers that characterise sporadic tasks running in a system. To harness this information, we rely on the formal notion of *simulation*:

Definition 16 Let $A = \langle V, E, S_0, F\rangle$ be an automaton. A *simulation relation* for A is a preorder $\succcurlyeq \subseteq V \times V$ s.t.:

1. For all S_1, S_2, S_3 s.t. $(S_1, S_2) \in E$ and $S_3 \succcurlyeq S_1$, there exists S_4 s.t. $(S_3, S_4) \in E$ and $S_4 \succcurlyeq S_2$.
2. For all S_1, S_2 s.t. $S_1 \succcurlyeq S_2$: $S_2 \in F$ implies $S_1 \in F$.

Whenever $S_1 \succcurlyeq S_2$, we say that S_1 *simulates* S_2. Whenever $S_1 \succcurlyeq S_2$ but $S_2 \not\succcurlyeq S_1$, we write $S_1 \succ S_2$.

Intuitively, this definition says that whenever a state S_3 simulates a state S_1, then S_3 can *mimic* every possible move of S_1 by moving to a similar state: for every edge (S_1, S_2), there is a corresponding edge (S_3, S_4), where S_4 simulates S_2. Moreover, we request that a *target state* can only be simulated by a target state. Remark that for a given automaton there can be several simulation relations (for instance, equality is always a simulation relation).

The key consequence of this definition is that **if** S_2 is a state that can reach F, and if $S_1 \succcurlyeq S_2$ **then** S_1 *can reach* F *too.* Indeed, if S_2 can reach F, there is a path $v_0, v_1, \ldots, v_n$ with $v_0 = S_2$ and $v_n \in F$. Using Definition 16 we can inductively build a path $v'_0, v'_1, \ldots, v'_n$ s.t. $v'_0 = S_1$ and $v'_i \succcurlyeq v_i$ for all $i \geqslant 0$. Thus, in particular $v'_n \succcurlyeq v_n \in F$, hence $v'_n \in F$ by Definition 16. This means that S_1 can reach F too. Thus, when we compute two states S_1 and S_2 with $S_1 \succcurlyeq S_2$, at some step of Algorithm 1, we *do not need to further explore the successors of* S_2. Indeed, Algorithm 1 tries to detect reachable target states. So, if S_2 cannot reach a failure state, it is safe not to explore its successors. Otherwise, if S_2 *can* reach a target state, then S_1 can reach a target state too, so it is safe to explore the successors of S_1 only. By exploiting this heuristic, Algorithm 1 could explore only a (small) subset of the states of A, which has the potential for a dramatic improvement in computation time. Remark that such techniques have already been exploited in the setting of *formal verification*, where several so-called *antichains algorithms* have been studied [9,11,13] and have proved to be *several order of magnitudes more efficient* than the classical techniques of the literature.

Formally, for a set of states $V' \subseteq V$, we let $\mathsf{Max}^{\succcurlyeq}\left(V'\right) = \{S \in V' \mid \nexists S' \in V' \text{ with } S' \succ S\}$. Intuitively, $\mathsf{Max}^{\succcurlyeq}\left(V'\right)$ is obtained from V' by removing all the states that are simulated by some other state in V'. So the states we keep in $\mathsf{Max}^{\succcurlyeq}\left(V'\right)$ are not comparable according to $\succcurlyeq$. Remark that such sets of incomparable elements are called *antichains*:

Definition 17 (Antichain) Given a set S and a partial order $\succcurlyeq$ on S, an *antichain* on S is a set $S' \subseteq S$ s.t. for all $s_1, s_2 \in S'$: $s_1 \not\succcurlyeq s_2$.

Then, we consider Algorithm 2 which is an improved version of Algorithm 1.

Algorithm 2: Improved breadth-first traversal.

```
1  begin
2      i ← 0 ;
3      R̃_0 ← S_0 ;
4      repeat
5          i ← i + 1 ;
6          R̃_i ← R̃_{i−1} ∪ Succ(R̃_{i−1}) ;
7          R̃_i ← Max^≽(R̃_i) ;
8          if R̃_i ∩ F ≠ ∅ then return Reachable ;
9      until R̃_i = R̃_{i−1};
10     return Not reachable ;
```

Proving the correctness and termination of Algorithm 2 is a little bit more involved than for Algorithm 1. We first prove the following ancillary result:

Lemma 5 $\mathsf{Max}^{\succcurlyeq}\left(\mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}(B)\right)\right) = \mathsf{Max}^{\succcurlyeq}(\mathsf{Succ}(B))$.

Proof We first show that $\mathsf{Max}^{\succcurlyeq}\left(\mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}(B)\right)\right) \subseteq \mathsf{Max}^{\succcurlyeq}(\mathsf{Succ}(B))$. By definition of $\mathsf{Max}^{\succcurlyeq}(B)$, we know that $\mathsf{Max}^{\succcurlyeq}(B) \subseteq A$. Moreover, Succ and $\mathsf{Max}^{\succcurlyeq}$ are monotonic w.r.t. set inclusion. Hence:

$$\begin{aligned} & \mathsf{Max}^{\succcurlyeq}(B) \subseteq B \\ \Rightarrow\ & \mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}(B)\right) \subseteq \mathsf{Succ}(B) \\ \Rightarrow\ & \mathsf{Max}^{\succcurlyeq}\left(\mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}(B)\right)\right) \subseteq \mathsf{Max}^{\succcurlyeq}(\mathsf{Succ}(B)) \end{aligned}$$

Then, we show that $\mathsf{Max}^{\succcurlyeq}\left(\mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}(B)\right)\right) \supseteq \mathsf{Max}^{\succcurlyeq}(\mathsf{Succ}(B))$. Let S_2 be a state in $\mathsf{Max}^{\succcurlyeq}(\mathsf{Succ}(B))$. Let $S_1 \in B$ be a state s.t. $(S_1, S_2) \in E$. Since, $S_2 \in \mathsf{Succ}(B)$, S_1 always exists. Since $S_1 \in B$, there exists $S_3 \in \mathsf{Max}^{\succcurlyeq}(B)$ s.t. $S_3 \succcurlyeq S_1$. By Definition 16, there is $S_4 \in \mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}(B)\right)$ s.t. $S_4 \succcurlyeq S_2$. To conclude, let us show *per absurdum* that S_4 is maximal in $\mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}(B)\right)$. Assume there exists $S_5 \in \mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}(B)\right)$ s.t. $S_5 \succ S_4$. Since $\mathsf{Max}^{\succcurlyeq}(B) \subseteq A$, S_5 is in $\mathsf{Succ}(B)$ too. Moreover, since $S_4 \succcurlyeq S_2$ and $S_5 \succ S_4$, we conclude that $S_5 \succ S_2$. Thus, there is, in $\mathsf{Succ}(B)$ and element $S_5 \succ S_2$. This contradict our hypothesis that $S_2 \in \mathsf{Max}^{\succcurlyeq}(\mathsf{Succ}(B))$.

□

Thanks to Lemma 5, we can establish a relationship between the sets R_i and $\widetilde{R}_i$ computed respectively by Algorithm 1 and Algorithm 2. This will help us in establishing the correctness of Algorithm 2 (see Theorem 2) hereunder:

Lemma 6 *Let A be an automaton and let $\succcurlyeq$ be a simulation relation for A. Let $R_0, R_1, \ldots$ and $\widetilde{R}_0, \widetilde{R}_1, \ldots$ denote respectively the sequence of sets computed by Algorithm 1 and Algorithm 2 on A. Then, for all $i \geqslant 0$: $\widetilde{R}_i = \mathsf{Max}^{\succcurlyeq}(R_i)$.*

Proof The proof is by induction on i. We first observe that for any pair of sets B and C, the following holds:

$$\begin{aligned} &\mathsf{Max}^{\succcurlyeq}\,(B \cup C) \\ =\ &\mathsf{Max}^{\succcurlyeq}\left(\mathsf{Max}^{\succcurlyeq}\,(B) \cup \mathsf{Max}^{\succcurlyeq}\,(C)\right) \end{aligned} \tag{1}$$

Base case $i = 0$ Clearly, $\mathsf{Max}^{\succcurlyeq}\,(R_0) = R_0$ since R_0 is a singleton. By definition $\widetilde{R}_0 = R_0$.

Inductive case $i = k$ *Induction hypothesis* we assume that $\widetilde{R}_{k-1} = \mathsf{Max}^{\succcurlyeq}\,(R_k)$. Then:

$$\begin{aligned} &\widetilde{R}_k \\ =\ &\mathsf{Max}^{\succcurlyeq}\left(\widetilde{R}_{k-1} \cup \mathsf{Succ}\left(\widetilde{R}_{k-1}\right)\right) && \text{By def.} \\ =\ &\mathsf{Max}^{\succcurlyeq}\left(\mathsf{Max}^{\succcurlyeq}\left(\widetilde{R}_{k-1}\right) \cup \mathsf{Max}^{\succcurlyeq}\left(\mathsf{Succ}\left(\widetilde{R}_{k-1}\right)\right)\right) && \text{by (1)} \\ =\ &\mathsf{Max}^{\succcurlyeq}\left(\mathsf{Max}^{\succcurlyeq}\left(\mathsf{Max}^{\succcurlyeq}\,(R_{k-1})\right) \cup \mathsf{Max}^{\succcurlyeq}\left(\mathsf{Succ}\left(\mathsf{Max}^{\succcurlyeq}\,(R_{k-1})\right)\right)\right) && \text{By I.H.} \\ =\ &\mathsf{Max}^{\succcurlyeq}\left(\mathsf{Max}^{\succcurlyeq}\,(R_{k-1}) \cup \mathsf{Max}^{\succcurlyeq}\,(\mathsf{Succ}\,(R_{k-1}))\right) && \text{By Lemma 5} \\ =\ &\mathsf{Max}^{\succcurlyeq}\,(R_{k-1} \cup \mathsf{Succ}\,(R_{k-1})) && \text{By (1)} \\ =\ &\mathsf{Max}^{\succcurlyeq}\,(R_k) && \text{By def.} \end{aligned}$$

□

Intuitively, this means that some state S that is in R_i could not be present in $\widetilde{R}_i$, but that we always keep in $\widetilde{R}_i$ a state S' that simulates S. Then, we can prove that:

Theorem 2 *For all automata $A = \langle V, E, S_0, T\rangle$, Algorithm 2 terminates and returns "`Reachable`" iff T is reachable in A.*

Proof The proof relies on the comparison between the sequence of sets $R_0, R_1, \ldots$ computed by Algorithm 1 (which is correct and terminates) and the sequence $\widetilde{R}_0, \widetilde{R}_1, \ldots$ computed by Algorithm 2.

Assume F is reachable in A in k steps and not reachable in less than k steps. Then, there exists a path $v_0, v_1, \ldots v_k$ with $v_0 \in S_0$, $v_k \in F$, and, for all $0 \leqslant i \leqslant k$ $v_i \in R_k$. Let us first show *per absurdum* that the loop in Algorithm 2 does not finish before the kth step. Assume it is not the case, i.e. there exists $0 < \ell < k$ s.t. $\widetilde{R}_\ell = \widetilde{R}_{\ell-1}$. This implies that $\mathsf{Max}^{\succcurlyeq}\,(R_\ell) = \mathsf{Max}^{\succcurlyeq}\,(R_{\ell-1})$ through Lemma 6. Since $R_\ell \neq R_{\ell-1}$, we deduce that all the states that have been added to R_ℓ are simulated by some state already present in $R_{\ell-1}$: for all $S \in R_\ell$, there is $S' \in R_{\ell-1}$ s.t. $S' \succcurlyeq S$. Thus, in particular, there is $S' \in R_{\ell-1}$ s.t. $S' \succcurlyeq v_\ell$. We consider two cases. Either there is $S' \in R_{\ell-1}$ s.t. $S' \succcurlyeq v_k$. Since $v_k \in F$, $F \cap R_{\ell-1} \neq \emptyset$, which contradicts our hypothesis that F is not reachable in less than k steps. Otherwise, let $0 \leqslant m < k$ be the least position in the path s.t. there is $S' \in R_{\ell-1}$ with $S' \succcurlyeq v_m$, but there is no $S'' \in R_{\ell-1}$ with $S'' \succcurlyeq v_{m+1}$. In this case, since $S' \succcurlyeq v_m$ and $(v_m, v_{m+1}) \in E$, there is $S \in \mathsf{Succ}\left(S'\right) \subseteq R_\ell$ s.t. $S \succcurlyeq v_{m+1}$. However, we have made the hypothesis that every element in R_ℓ is simulated by some element in $R_{\ell-1}$. Thus, there is $S'' \in R_{\ell-1}$ s.t. $S'' \succcurlyeq S$. Since $S \succcurlyeq v_{m+1}$, we deduce that $S'' \succcurlyeq v_{m+1}$, with $S'' \in R_{\ell-1}$, which contradicts our assumption that $S'' \notin R_{\ell-1}$. Thus, Algorithm 2 will not stop before the kth iteration, and we know that there is $S_F \in R_k$ s.t. $S_F \in F$. By Lemma 6, $\widetilde{R}_k = \mathsf{Max}^{\succcurlyeq}\,(R_k)$, hence there is $S' \in \widetilde{R}_k$ s.t. $S' \succcurlyeq S$. By Definition 16, $S' \in F$ since $S \in F$. Hence, $\widetilde{R}_k \cap F \neq \emptyset$ and Algorithm 2 terminates after k steps with the correct answer.

Otherwise, assume F is not reachable in A. Hence, for every $i \geqslant 0$, $R_i \cap F = \emptyset$. Since $\widetilde{R}_i \subseteq R_i$ for all $i \geqslant 0$, we conclude that $\widetilde{R}_i \cap F = \emptyset$ for all $i \geqslant 0$. Hence, Algorithm 2 never returns "`Reachable`" in this case. It remains to show that the **repeat**

loop eventually terminates. Since F is not reachable in A, there is k s.t. $R_k = R_{k-1}$. Hence, $\mathsf{Max}^{\succcurlyeq}(R_k) = \mathsf{Max}^{\succcurlyeq}(R_{k-1})$. By Lemma 6 this implies that $\widetilde{R}_k = \widetilde{R}_{k-1}$. Thus, Algorithm 2 finishes after k steps and returns **"`Not reachable`"**.

□

In order to apply Algorithm 2, it remains to show how to compute a simulation relation, which should contain as many pairs of states as possible, since this raises the chances to avoid exploring some states during the breadth-first search. It is well-known that the largest simulation relation of an automaton can be computed in polynomial time w.r.t. the size of the automaton [20]. However, this requires first computing the whole automaton, which is exactly what we want to avoid in our case. So we need to define simulations relations that can be computed *a priori*, only by considering the structure of the states (in our case, the functions nat and rct). This is the purpose of the next section.

5 Idle tasks simulation relation

In this section we define a simulation relation $\succcurlyeq_{idle}$, called the *idle tasks simulation relation* that can be computed by inspecting the nat and rct values stored in the states.

Definition 18 Let τ be a set of sporadic tasks. Then, the *idle tasks preorder* $\succcurlyeq_{idle} \subseteq \mathsf{States}(\tau) \times \mathsf{States}(\tau)$ is s.t. for all S_1,S_2: $S_1 \succcurlyeq_{idle} S_2$ iff

1. $\mathrm{rct}_{S_1} = \mathrm{rct}_{S_2}$;
2. for all τ_i s.t. $\mathrm{rct}_{S_1}(\tau_i) = 0$: $\mathrm{nat}_{S_1}(\tau_i) \leqslant \mathrm{nat}_{S_2}(\tau_i)$;
3. for all τ_i s.t. $\mathrm{rct}_{S_1}(\tau_i) > 0$: $\mathrm{nat}_{S_1}(\tau_i) = \mathrm{nat}_{S_2}(\tau_i)$.

Notice the relation is reflexive as well as transitive, and thus indeed a preorder. It also defines a partial order on $\mathsf{States}(\tau)$ as it is antisymmetric. Moreover, since $S_1 \succcurlyeq_{idle} S_2$ implies that $\mathrm{rct}_{S_1} = \mathrm{rct}_{S_2}$, we also have $\mathsf{Active}(S_1) = \mathsf{Active}(S_2)$. Intuitively, a state S_1 simulates a state S_2 iff (i) S_1 and S_2 coincide on all the active tasks (i.e., the tasks τ_i s.t. $\mathrm{rct}_{S_1}(\tau_i) > 0$), and (ii) the nat of each idle task is not larger in S_1 than in S_2.

Lemma 7 *Let S_1 and S_2 be two states such that $S_1 \succcurlyeq_{idle} S_2$. Then,* $\mathsf{Eligible}(S_1) \supseteq \mathsf{Eligible}(S_2)$.

Proof Let τ_i be a task in $\mathsf{Eligible}(S_2)$ and let us show that $\tau_i \in \mathsf{Eligible}(S_1)$. Since $\tau_i \in \mathsf{Eligible}(S_2)$, $\mathrm{rct}_{S_2}(\tau_i) = 0$ and $\mathrm{nat}_{S_2}(\tau_i) \leqslant 0$, by definition of $\mathsf{Eligible}$. Thus, since $S_1 \succcurlyeq_{idle} S_2$, $\mathrm{rct}_{S_1}(\tau_i) = \mathrm{rct}_{S_2}(\tau_i) = 0$, and $\mathrm{nat}_{S_1}(\tau_i) \leqslant \mathrm{nat}_{S_2}(\tau_i) \leqslant 0$, by definition of $\succcurlyeq_{idle}$. Hence $\tau_i \in \mathsf{Eligible}(S_1)$. □

Definition 19 Let S_1 and S_2 be two states such that $S_1 \succcurlyeq_{idle} S_2$, let $\tau' \subseteq \mathsf{Eligible}(S_2)$. Then[10], for all $\overline{S}_2 \in S_2^{\tau'}$, a state $\widetilde{S}_1 \in S_1^{\tau'}$ is called a *τ'-idle analog* of $\overline{S}_2$ iff:

$$\forall \tau_i \in \tau' \cdot \mathrm{nat}_{\widetilde{S}_1}(\tau_i) = \mathrm{nat}_{\overline{S}_2}(\tau_i)$$

Lemma 8 *Let S_1 and S_2 be two states s.t. $S_1 \succcurlyeq_{idle} S_2$ and let $\tau' \subseteq \mathsf{Eligible}(S_2)$ be a subset of the task set. Then, for all $\overline{S}_2 \in S_2^{\tau'}$ there is a τ'-idle analog $\widetilde{S}_1 \in S_1^{\tau'}$ of $\overline{S}_2$*

[10] Recall that S^{τ} denotes the set of τ-request successors of S, see Definition 12.

Proof Definitions 12 and 18 tell us that for each $\tau_i \in \tau' \subseteq \mathsf{Eligible}(S_2)$, we have $\mathrm{nat}_{S_1}(\tau_i) \leqslant \mathrm{nat}_{S_2}(\tau_i) \leqslant 0$ and $\mathrm{rct}_{S_1}(\tau_i) = \mathrm{rct}_{S_2}(\tau_i) = 0$. By definition, any $\overline{S}_2 \in S_2^{\tau'}$ satisfies the following:

$$\forall \tau_i \in \tau' : \mathrm{nat}_{\overline{S}_2}(\tau_i) \in I_2^i = [\mathrm{nat}_{S_2}(\tau_i) + T_i, T_i]$$

Similarly, by Definition 12, for all vector $(k_1, k_2, \ldots, k_n)$ (where $n = |\tau'|$) s.t. for all $1 \leqslant i \leqslant n$: $k_i \in I_1^i = [\mathrm{nat}_{S_1}(\tau_i) + T_i, T_i]$, there exists $\overline{S}_1 \in S_1^{\tau'}$ with $\mathrm{nat}_{\overline{S}_1}(\tau_i) = k_i$ for all $1 \leqslant i \leqslant n$. That is, each time we fix a vector $(k_1, k_2, \ldots, k_n)$ of values in $I_1^1 \times I_1^2 \times \cdots \times I_1^n$, we can find a a state $\overline{S}_1 \in S_1^{\tau'}$ s.t. the nat of each task τ_i is exactly k_i. However, $\mathrm{nat}_{S_1}(\tau_i) \leqslant \mathrm{nat}_{S_2}(\tau_i)$ for all $1 \leqslant i \leqslant n$ and we conclude that $\mathrm{nat}_{\overline{S}_2}(\tau_i) \in I_2^i \subseteq I_1^i$ for all $1 \leqslant i \leqslant n$. Thus, the vector $\big(\mathrm{nat}_{\overline{S}_2}(\tau_1), \mathrm{nat}_{\overline{S}_2}(\tau_2), \ldots, \mathrm{nat}_{\overline{S}_2}(\tau_n)\big)$ is in $I_1^1 \times I_1^2 \times \cdots \times I_1^n$. Hence, there exists $\widetilde{S}_1 \in S_1^{\tau'}$ s.t. for all $1 \leqslant i \leqslant n$: $\mathrm{nat}_{\widetilde{S}_1}(\tau_i) = \mathrm{nat}_{\overline{S}_2}(\tau_i)$. □

Let us show that $\succcurlyeq_{idle}$ is indeed a simulation relation when we consider a *memoryless* scheduler (which is often the case in practice):

Theorem 3 *Let τ be a set of sporadic tasks and let* Run *be a* memoryless *(deterministic) scheduler for τ on m processors. Then, $\succcurlyeq_{idle}$ is a simulation relation for $A(\tau, \mathsf{Run})$.*

Proof Let S_1, S_1' and S_2 be three states in $\mathsf{States}(\tau)$ s.t. $(S_1, S_1') \in E$ and $S_2 \succcurlyeq_{idle} S_1$, and let us show that there exists $S_2' \in \mathsf{States}(\tau)$ with $(S_2, S_2') \in E$ and $S_2' \succcurlyeq_{idle} S_1'$.

Since $(S_1, S_1') \in E$, there exists $\overline{S}_1$ and $\tau' \subseteq \mathsf{Eligible}(S_1)$ s.t. $S_1 \xrightarrow{\tau'} \overline{S}_1 \xrightarrow{\mathsf{Run}} S_1'$, by Definition 14. Let $\overline{S}_2$ be a τ'-idle analog state of $\overline{S}_1$ (which we know exists through Lemma 8) and let us show that $\overline{S}_2 \succcurlyeq_{idle} \overline{S}_1$:

1. for all $\tau_i \in \tau'$: $\mathrm{rct}_{\overline{S}_1}(\tau_i) = C_i = \mathrm{rct}_{\overline{S}_2}(\tau_i)$. For all $\tau_i \notin \tau'$: $\mathrm{rct}_{\overline{S}_1}(\tau_i) = \mathrm{rct}_{S_1}(\tau_i)$, $\mathrm{rct}_{\overline{S}_2}(\tau_i) = \mathrm{rct}_{S_2}(\tau_i)$, and, since $S_2 \succcurlyeq_{idle} S_1$: $\mathrm{rct}_{S_1}(\tau_i) = \mathrm{rct}_{S_2}(\tau_i)$. Thus, $\mathrm{rct}_{\overline{S}_1} = \mathrm{rct}_{\overline{S}_2}$.
2. For all $\tau_i \in \tau'$, $\mathrm{nat}_{\overline{S}_1}(\tau_i) = \mathrm{nat}_{\overline{S}_2}(\tau_i)$, as $\overline{S}_2$ is the τ'-idle analog of $\overline{S}_1$.
3. For all $\tau_i \notin \tau'$:

$$\mathrm{nat}_{\overline{S}_1}(\tau_i) = \mathrm{nat}_{S_1}(\tau_i) \text{ and } \mathrm{nat}_{\overline{S}_2}(\tau_i) = \mathrm{nat}_{S_2}(\tau_i) \tag{2}$$

and

$$\mathrm{rct}_{\overline{S}_1}(\tau_i) = \mathrm{rct}_{S_1}(\tau_i) \text{ and } \mathrm{rct}_{\overline{S}_2}(\tau_i) = \mathrm{rct}_{S_2}(\tau_i) \tag{3}$$

by Definition 12. We consider two further cases:

(a) If $\mathrm{rct}_{\overline{S}_1}(\tau_i) = 0$, then $\mathrm{rct}_{\overline{S}_2}(\tau_i) = 0$, by point 1 above. Hence, by (3), $\mathrm{rct}_{S_1}(\tau_i) = 0$ and $\mathrm{rct}_{S_2}(\tau_i) = 0$. Thus, since $S_2 \succcurlyeq_{idle} S_1$: $\mathrm{nat}_{S_1}(\tau_i) \leqslant \mathrm{nat}_{S_2}(\tau_i)$, by Definition 18. Hence, $\mathrm{nat}_{\overline{S}_1}(\tau_i) \leqslant \mathrm{nat}_{\overline{S}_2}(\tau_i)$, by (2).

(b) If $\mathrm{rct}\,\overline{S}_1(\tau_i) > 0$, then $\mathrm{rct}_{\overline{S}_2}(\tau_i) > 0$, by point 1 above. Hence, by (3), $\mathrm{rct}_{S_1}(\tau_i) > 0$ and $\mathrm{rct}_{S_2}(\tau_i) > 0$. Thus, since $S_2 \succcurlyeq_{idle} S_1$: $\mathrm{nat}_{S_1}(\tau_i) = \mathrm{nat}_{S_2}(\tau_i)$, by Definition 18. Hence, $\mathrm{nat}_{\overline{S}_1}(\tau_i) = \mathrm{nat}_{\overline{S}_2}(\tau_i)$, by (2).

We conclude that, for all task τ_i, either $\mathrm{nat}_{\overline{S}_1}(\tau_i) = \mathrm{nat}_{\overline{S}_2}(\tau_i)$, or $\mathrm{rct}\,\overline{S}_1(\tau_i) = \mathrm{rct}\,\overline{S}_2(\tau_i) = 0$ and $\mathrm{nat}_{\overline{S}_1}(\tau_i) \leqslant \mathrm{nat}_{\overline{S}_2}(\tau_i)$.

Hence, we conclude that $\overline{S}_2 \succcurlyeq_{idle} \overline{S}_1$. Then observe that, by Definition 16, $\overline{S}_2 \succcurlyeq_{idle} \overline{S}_1$ implies that $\mathsf{Active}(\overline{S}_1) = \mathsf{Active}(\overline{S}_2)$. Let τ_i be a task in $\mathsf{Active}(\overline{S}_1)$, hence $\mathrm{rct}_{\overline{S}_1}(\tau_i) > 0$. In this case, and since $\overline{S}_2 \succcurlyeq_{idle} \overline{S}_1$, we conclude that $\mathrm{rct}_{\overline{S}_1}(\tau_i) = \mathrm{rct}_{\overline{S}_2}(\tau_i)$ and $\mathrm{nat}_{\overline{S}_1}(\tau_i) = \mathrm{nat}_{\overline{S}_2}(\tau_i)$ for all $\tau_i \in \mathsf{Active}(\overline{S}_1) = \mathsf{Active}(\overline{S}_2)$. Thus, since Run is memoryless by hypothesis, $\mathsf{Run}(\overline{S}_1) = \mathsf{Run}(\overline{S}_2)$, by Definition 8. Let S_2' be the unique state s.t. $\overline{S}_2 \xrightarrow{\mathsf{Run}} S_2'$, and let us show that $S_2' \succcurlyeq_{idle} S_1'$:

1. Since $\overline{S}_2 \succcurlyeq_{idle} \overline{S}_1$, we know that $\mathrm{rct}_{\overline{S}_1} = \mathrm{rct}_{\overline{S}_2}$. Let τ_i be a task in $\mathsf{Run}(\overline{S}_1) = \mathsf{Run}(\overline{S}_2)$. By Definition 11: $\mathrm{rct}_{S_1'}(\tau_i) = \mathrm{rct}_{\overline{S}_1}(\tau_i) - 1$ and $\mathrm{rct}_{S_2'}(\tau_i) = \mathrm{rct}_{\overline{S}_2}(\tau_i) - 1$. Hence, $\mathrm{rct}_{S_1'}(\tau_i) = \mathrm{rct}_{S_2'}(\tau_i)$. For a task $\tau_i \notin \mathsf{Run}(\overline{S}_1) = \mathsf{Run}(\overline{S}_2)$, we have $\mathrm{rct}_{S_1'}(\tau_i) = \mathrm{rct}_{\overline{S}_1}(\tau_i)$ and $\mathrm{rct}_{S_2'}(\tau_i) = \mathrm{rct}_{\overline{S}_2}(\tau_i)$, again by Definition 11. Hence, $\mathrm{rct}_{S_1'}(\tau_i) = \mathrm{rct}_{S_2'}(\tau_i)$. We conclude that $\mathrm{rct}_{S_1'} = \mathrm{rct}_{S_2'}$.
2. Let τ_i be a task s.t. $\mathrm{rct}_{S_1'}(\tau_i) = 0$. By Definition 11: $\mathrm{nat}_{S_1'}(\tau_i) = \max\{0, \mathrm{nat}_{\overline{S}_1}(\tau_i) - 1\}$ and $\mathrm{nat}_{S_2'}(\tau_i) = \max\{0, \mathrm{nat}_{\overline{S}_2}(\tau_i) - 1\}$. However, since $\overline{S}_2 \succcurlyeq_{idle} \overline{S}_1$, we know that $\mathrm{nat}_{\overline{S}_1}(\tau_i) \leqslant \mathrm{nat}_{\overline{S}_2}(\tau_i)$. We conclude that $\mathrm{nat}_{S_1'}(\tau_i) \leqslant \mathrm{nat}_{S_2'}(\tau_i)$.
3. Let τ_i be a task s.t. $\mathrm{rct}_{S_1'}(\tau_i) > 0$. By Definition 11: $\mathrm{nat}_{S_1'}(\tau_i) = \mathrm{nat}_{\overline{S}_1}(\tau_i) - 1$ and $\mathrm{nat}_{S_2'}(\tau_i) = \mathrm{nat}_{\overline{S}_2}(\tau_i) - 1$. Since $\mathrm{rct}_{S_1'}(\tau_i) > 0$, we have $\mathrm{rct}_{\overline{S}_1}(\tau_i) > 0$ too, since rct can only decrease with time elapsing. Since $S_1 \succcurlyeq_{idle} S_2$ we have also $\mathrm{nat}_{\overline{S}_2}(\tau_i) = \mathrm{nat}_{\overline{S}_1}(\tau_i)$. We conclude that $\mathrm{nat}_{S_1'}(\tau_i) = \mathrm{nat}_{S_2'}(\tau_i)$.

To conclude the proof it remains to show that, if $S_2 \succcurlyeq_{idle} S_1$ and $S_1 \in \mathsf{Fail}_\tau$ then $S_2 \in \mathsf{Fail}_\tau$ too. Let τ_i be a task s.t $\mathrm{laxity}_{S_1}(\tau_i) = \mathrm{nat}_{S_1}(\tau_i) - (T_i - D_i) - \mathrm{rct}_{S_1}(\tau_i) < 0$. Since $S_2 \succcurlyeq_{idle} S_1$: $\mathrm{rct}_{S_2}(\tau_i) = \mathrm{rct}_{S_1}(\tau_i)$, and $\mathrm{nat}_{S_2}(\tau_i) \leqslant \mathrm{nat}_{S_1}(\tau_i)$. Hence, $\mathrm{laxity}_{S_2}(\tau_i) = \mathrm{nat}_{S_2}(\tau_i) - (T_i - D_i) - \mathrm{rct}_{S_2}(\tau_i) \leqslant \mathrm{laxity}_{S_1}(\tau_i) < 0$, and thus, $S_2 \in \mathsf{Fail}_\tau$. □

Note that Theorem 3 does *not* require the scheduler to be work-conserving. Theorem 3 tells us that any state where tasks have to wait until their next job release can be simulated by a corresponding state where they can release their job earlier, regardless of the specifics of the scheduling policy as long as it is deterministic, sustainable and memoryless, which is what many popular schedulers are in practice, such as preemptive DM or EDF.

Fig. 9 illustrates the effect of using $\succcurlyeq_{idle}$ with Algorithm 2. If a state S_1 has been reached previously and we find another state S_2 such that $S_1 \succcurlyeq_{idle} S_2$, then we can avoid exploring S_2 and its successors altogether. However, note that this does not mean we will never reach a successor of S_2 as they may be reached through other paths (or indeed, may have been reached already).

6 Experimental results

We implemented both Algorithm 1 (denoted *BF*) and Algorithm 2 (denoted *ACBF* for "antichain breadth-first") in C++ using the STL and Boost libraries 1.48.0. We ran head-to-head tests on a Mac Pro system equipped with a six-core 3.33 GHz Intel Xeon processor and 32 GB of RAM running under Mac OS X 10.7.4. Our programs were compiled with Apple Inc.'s distribution of g++ 4.2.1 with flags for maximal optimization.

Note that in all our experiments described hereafter, intermediary τ'-request states are counted as well, even though they are not effectively present in the state space of our

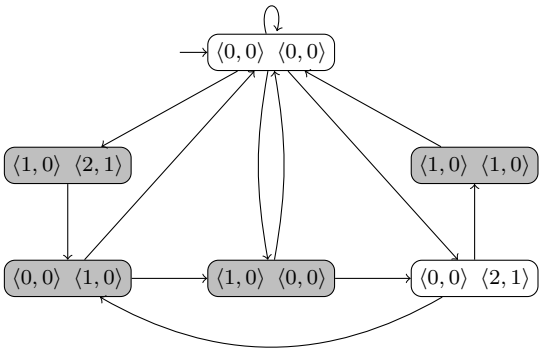


Fig. 9 Algorithm 2 exploits simulation relations to avoid exploring states needlessly. With $\succcurlyeq_{idle}$ on this small example, all grey states can be avoided as they are simulated by another state. For instance, $\big(\langle 0,0\rangle, \langle 2,1\rangle\big) \succcurlyeq_{idle} \big(\langle 1,0\rangle, \langle 2,1\rangle\big)$ and $\big(\langle 0,0\rangle, \langle 0,0\rangle\big) \succcurlyeq_{idle} \big(\langle 1,0\rangle, \langle 1,0\rangle\big)$.

automaton defined in Section 2. We argue that this is not an issue when comparing BF and ACBF, as removing those states from the count would not cause affect significantly ratios of states explored by the algorithms. Indeed, the number of τ'-request successors explored is tightly linked to the number of clock-tick successors explored.

We based our experimental protocol on that used in [3]. We generated random task sets where task minimum interarrival times T_i were uniformly distributed in $\{1, 2, \ldots, T_{\max}\}$, task WCETs C_i followed an exponential distribution of mean $0.35\,T_i$ and relative deadlines were uniformly distributed in $\{C_i, \ldots, T_i\}$ in the case of constrained deadlines, and in $\{C_i, \ldots, T_i, \ldots, 4T_i\}$ in the case of arbitrary deadlines. Task sets where $n \leqslant m$ were dropped as well as sets where $\sum_i C_i/T_i > m$. Duplicate task sets were discarded as were sets which could be scaled down by an integer factor. We used EDF as scheduler and simulated $m = 2$ for all experiments. Execution times (approximated by measuring wall time as measuring thread-level execution times proved very difficult on our experimental platform) were measured using the POSIX `gettimeofday()` primitive.

We ran several sets of experiments. The first and the second compare the performance of BF and ACBF on randomly generated sets of *constrained deadline* task sets. The third compare BF and ACBF on randomly generated sets of *arbitrary deadline* task sets. Then, an experiment explores the relationship between the performance of the ACBF algorithm and the size of the largest antichain built during the execution. Our last experiment draws a link between the number of tasks of the system and the number of (reachable) states in the Baker-Cirinei automaton, which is, in our opinion, a good measure of the *hardness* of a particular instance.

BF vs. ACBF, constrained deadlines Our first experiment used $T_{\max} = 6$ and we generated 940 constrained-deadline task sets following the previous rules (of which 383 were EDF-schedulable). Task sets contained between 3 and 7 tasks (for each number of task, 188 task sets were generated), with total utilization between 1 and 2. Fig. 10 showcases the performance of both algorithms on these sets. The number of states explored by BF before halting gives a notion of how big the automaton was (if no

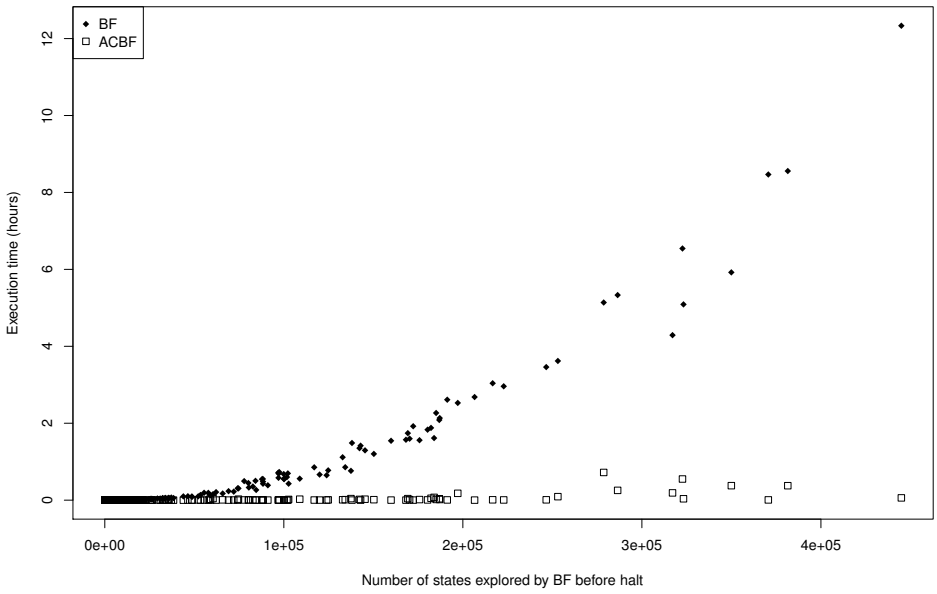


Fig. 10 States explored by BF before halt vs. execution time of BF and ACBF (940 task sets with $T_{\max} = 6$, constrained deadlines).

failure state is reachable, the number is exactly the number of states in the automaton that are reachable from the initial state; if a failure state is reachable, BF halts as soon as a failure state is found). It can be seen that while ACBF and BF show similar performance for fairly small systems, ACBF outperforms BF for larger systems, and we can thus conclude that the antichains technique *scales better*. The largest system analyzed in this experiment was schedulable (and BF thus had to explore it completely), contained 444,917 states and was handled in over 12 hours with BF, whereas ACBF clocked in at 4 minutes.

Fig. 11 shows, for the same experiment, a comparison between explored states by BF and ACBF. This comparison is more objective than the previous one, as it does not account for the actual efficiency of our crude implementations. As can be seen, the simulation relation allows ACBF to drop a considerable amount of states from its exploration as compared with BF: on average, 53.9% were avoided (37.4% in the case of unschedulable systems which cause an early halt, 77.9% in the case of schedulable systems). This of course largely explains the better performance of ACBF, but we must also take into account the overhead due to the more complex algorithm. In fact, we found that in some cases, ACBF would yield worse performance than BF. However, to the best of our knowledge, this only seems to occur in cases where BF took relatively little time to execute (less than five seconds) and is thus of no concern in practice.

Our second experiment used 5,000 randomly generated constrained-deadline task sets using $T_{\max} = 8$ (of which 3,175 were EDF-schedulable) and was intended to give

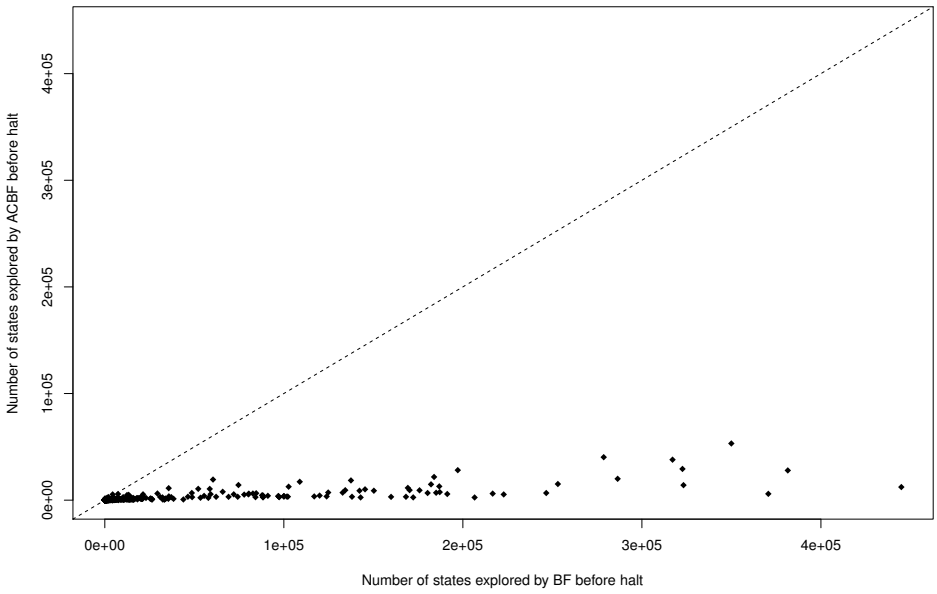


Fig. 11 States explored by BF before halt vs. states explored by ACBF before halt (940 task sets with $T_{\max} = 6$, constrained deadlines).

a rough idea of the limits of our current ACBF implementation. Task sets contained between 3 and 8 tasks with total utilization between 0.375 and 2. Fig. 12 plots the number of states explored by ACBF before halting versus its execution time. We can first notice the plot looks remarkably similar to BF in Fig. 10, which seems to confirm the exponential complexity of ACBF which we predicted. The largest schedulable system considered necessitated exploring 198,072 states and required roughly 5.5 hours. As a spot-check, we ran BF on a schedulable system where ACBF halted after exploring 14,754 states in 78 seconds; BF converged after just over 6 hours[11], exploring 434,086 states.

BF vs. ACBF, arbitrary deadlines Our third experiment used 800 randomly generated *arbitrary-deadline* task sets using $T_{\max} = 6$ (of which 782 were EDF-schedulable). Task sets contained between 3 and 6 tasks (for each number of tasks, 200 systems were generated), with total utilization between 1 and 2. Fig. 13 shows that under arbitrary deadlines, the performance of ACBF is much better than BF. It is also notable that the number of states explored by BF is at most an order of magnitude larger than under constrained deadlines. The largest system analyzed was schedulable and had 459,483 states. BF decided its schedulability in roughly 10.5 hours, whereas ACBF reached the

[11] Remark that this does not contradict the measured time of 12h on 444,000 states given above. Indeed, the running time of the algorithm depends on the number of states of the automaton, but also on the *branching degree* of the automaton's graph.

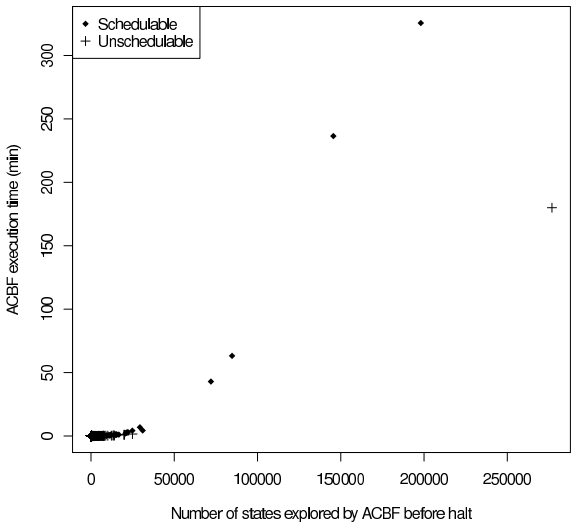


Fig. 12 States explored by ACBF before halt vs. ACBF execution time (5,000 task sets with $T_{\max} = 8$, constrained deadlines).

same conclusion in 1 hour and 45 minutes after exploring 165,815 states (corresponding to 36.1% of the automaton).

As for constrained deadlines, we also compared the number of states explored by BF and ACBF in each instance of our arbitrary-deadline task sets to give a more objective comparison, given in Fig. 14. On average, ACBF dropped roughly as many states as under constrained deadlines (72.9% globally, 74.1% for schedulable systems, albeit only 20.5% for unschedulable systems).

Fig. 15 plots the time performance of ACBF versus the number of states it explored before deciding schedulability.

Performance of ACBF vs. size of the largest antichain Since the main bottleneck in our ACBF implementation is state-by-state comparisons (under the idle tasks simulation relation) between the current antichain of states (i.e. known states that are minimal w.r.t. the simulation preorder) and the next fringe of successor states, we attempted to relate the performance of ACBF with the largest antichain used during exploration (in effect being the maximum size of the data structure we use to store states in our simulator). The result of this comparison is given in Fig. 16. In practice, we so far have to conclude that there is no trivial way to relate automaton size and the performance of the ACBF heuristic.

Number of tasks vs. size of the automaton Figure 17 illustrates an apparent exponential relation between the size of the automata and the number of tasks in the task set for a given $T_{\max}$. We observe that, for a given number of tasks, arbitrary deadline systems tend to get larger than constrained deadline systems. This may be explained by the

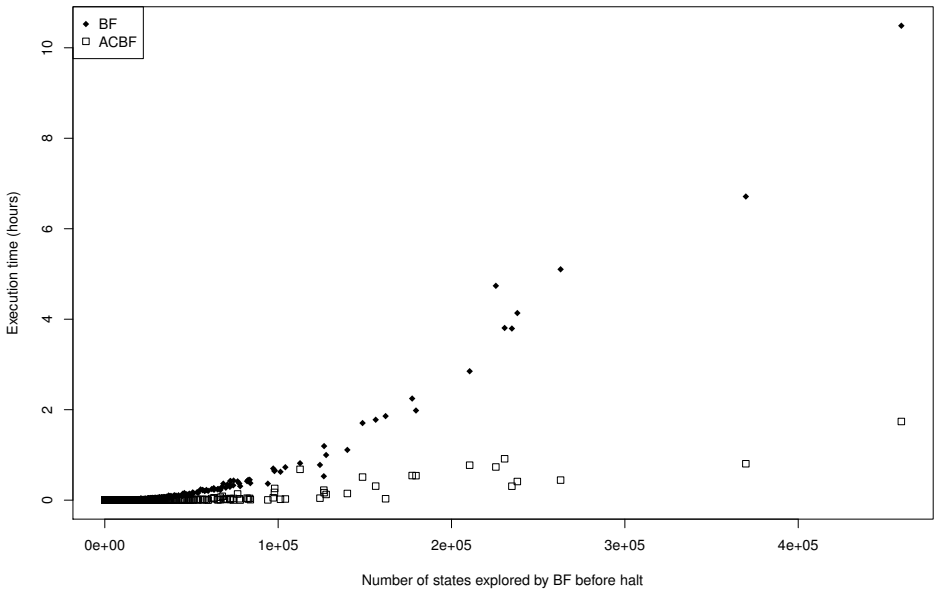


Fig. 13 States explored by BF before halt vs. execution time of BF and ACBF (1,000 task sets with $T_{\max} = 6$, arbitrary deadlines).

fact that nat values for a given task may have a larger range in arbitrary deadline systems.

Conclusion Our experimental results thus yield several interesting observations. The number of states explored by ACBF using the idle tasks simulation relation is significantly less on average than BF, both under constrained and arbitrary deadlines. This gives an objective metric to quantify the computational performance gains made by ACBF w.r.t. BF. In practice using our implementation, ACBF outperforms BF for any reasonably-sized automaton, but we have seen that while our current implementation of ACBF defeats BF, it gets slow itself for slightly more complicated task sets. However, we expect smarter implementations and more powerful simulation relations to push ACBF much further.

7 Conclusions and future work

In this work, we have studied the problem of scheduling a set of sporadic tasks on an identical multiprocessor platform. We have revisited the formalisation of this problem given by Baker and Cirinei [3], and have relied on this formalism to characterise precisely the complexity of the problem, which we prove to be PSPACE-complete.

Next, we have successfully adapted a novel algorithmic technique developed by the formal verification community, known as antichain algorithms [9, 11], to greatly improve

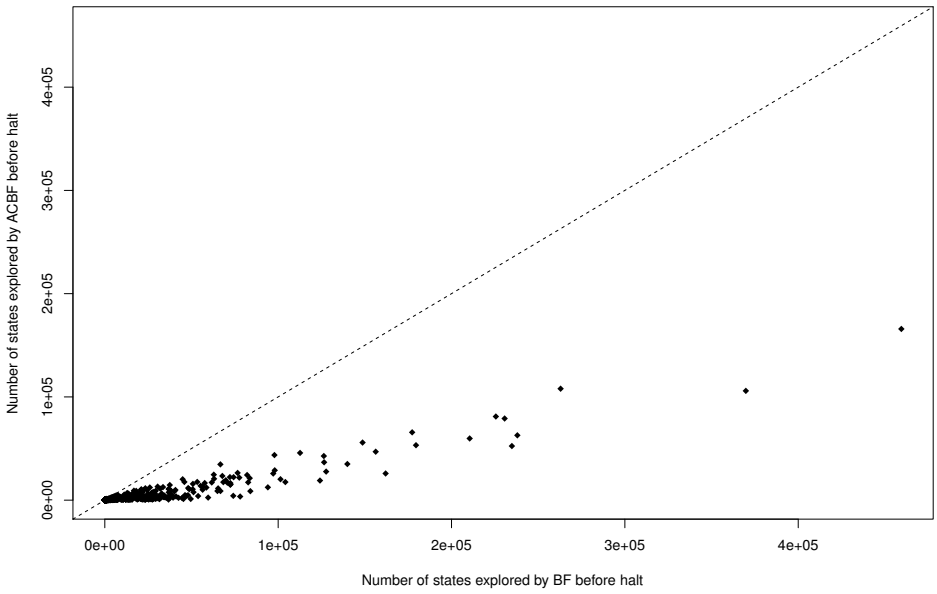


Fig. 14 States explored by BF before halt vs. states explored by ACBF before halt (1,000 task sets with $T_{\max} = 6$, arbitrary deadlines).

the performance of an existing exact schedulability test for sporadic hard real-time tasks on identical multiprocessor platforms [3]. To achieve this, we have developed and proved the correctness of a simulation relation on a formal model of the scheduling problem. While our algorithm has the same worst-case performance as a naive approach, we have shown experimentally that our preliminary implementation can still outperform the latter in practice.

The model introduced in Section 2 yields the added contribution of bringing a fully formalized description of the scheduling problem we considered. This allowed us to formally define various scheduling concepts such as memorylessness, work-conserving scheduling and various scheduling policies. These definitions are univocal and not open to interpretation, which we believe is an important consequence. We also clearly define what an execution of the system is, as any execution is a possibly infinite path in the automaton, and all possible executions are accounted for. For all these extensions, the precise complexity of the problem remains, as far as we know, an open question. Symmetrically, characterising the complexity of subcases of MSS (for instance, when the scheduler is assumed to be work-conserving) would certainly be interesting from a practical point of view.

The task model introduced in Section 2 can be further extended to enable study of more complex problems, such as job-level parallelism and semi-partitioned scheduling. The model introduced in Section 2 can also be extended to support broader classes of schedulers. This was briefly touched on in [3]. For example, storing the previous

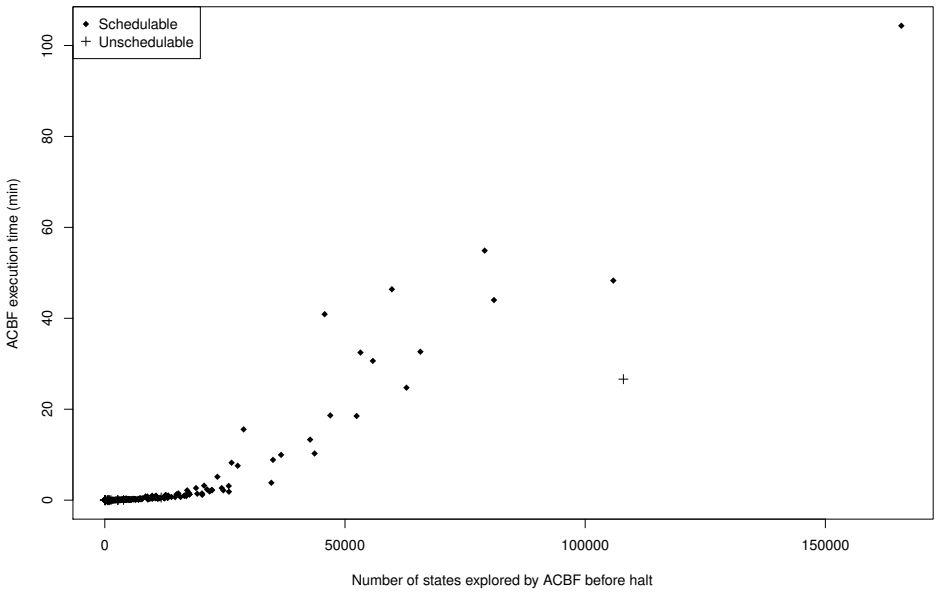


Fig. 15 States explored by ACBF before halt vs. ACBF execution time (1,000 task sets with $T_{\max} = 6$, arbitrary deadlines).

scheduling choice in each state would allow modelling of non-preemptive schedulers. As underlined in [15], it would also be interesting to consider modeling schedulers which are not FIFO but allow for task-level parallelism, since the two classes of schedulers are incomparable.

It has not yet been attempted to properly optimize our antichain algorithm by harnessing adequate data structures; our objective in this work was primarily to get a preliminary "proof-of-concept" comparison of the performance of the naive and antichain algorithms. Adequate implementation of structures such as *binary decision diagrams* [7] and *covering sharing trees* [10] should allow pushing the limits of the antichain algorithm's performance.

Antichain algorithms should terminate quicker by using coarser simulation preorders. Researching other simulation preorders on our model, particularly preorders that are a function of the chosen scheduling policy, is also key to improving performance. Determining the complexity class of sporadic task set feasibility on identical multiprocessor platforms is also of interest, as it may tell us whether other approaches could be used to solve the problem.

References

1. Abdeddaïm, Y., Maler, O.: Preemptive job-shop scheduling using stopwatch automata. In: AIPS-02 Workshop on Planning via Model-Checking, Toulouse, France, pp. 7–13 (2002)

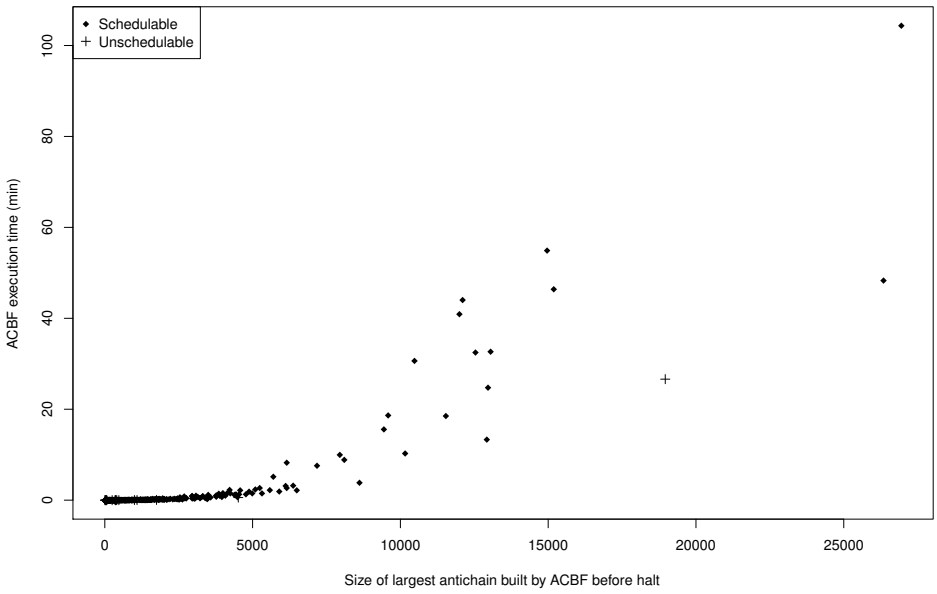
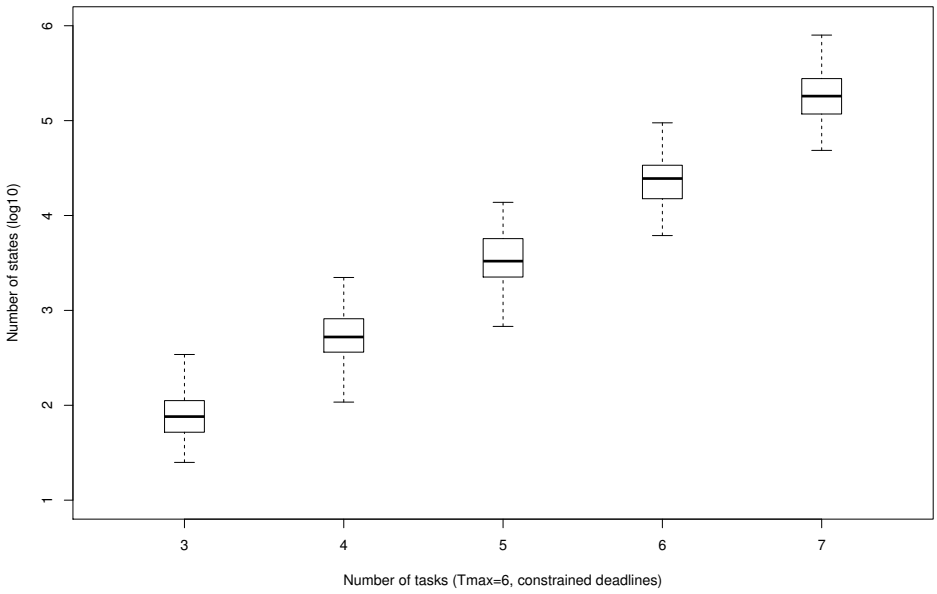
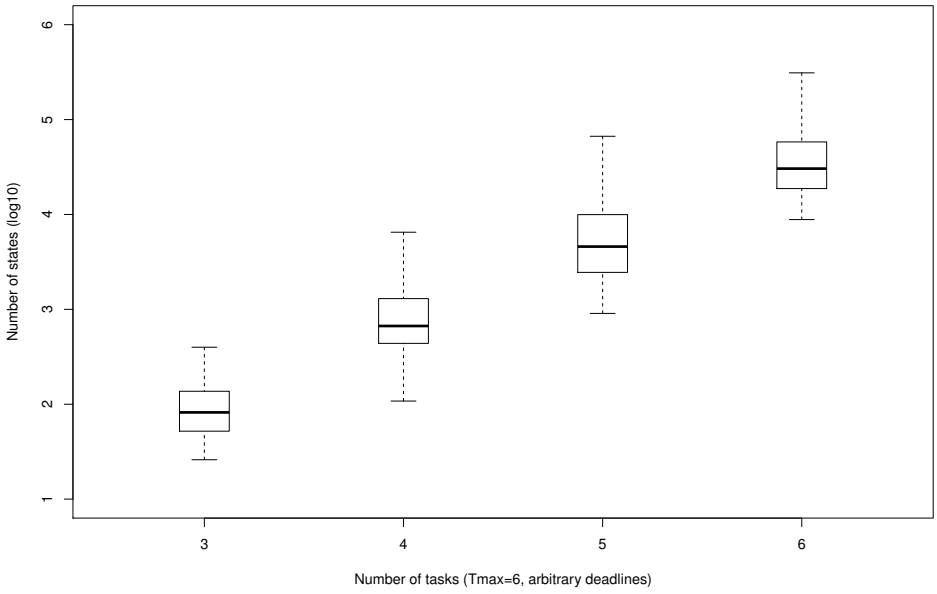


Fig. 16 Largest antichain built by ACBF before halt vs. ACBF execution time (1,000 task sets with $T_{\max} = 6$, arbitrary deadlines).

2. Baker, T.P., Baruah, S.K.: Schedulability analysis of multiprocessor sporadic task systems. In: I. Lee, J.Y.T. Leung, S. Son (eds.) Handbook of Real-Time and Embedded Systems. Chapman & Hall/CRC Press (2007)
3. Baker, T.P., Cirinei, M.: Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In: Tovar et al. [25], pp. 62–75
4. Baruah, S.K., Fisher, N.: Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems. In: Tovar et al. [25], pp. 204–216
5. Bertogna, M., Baruah, S.K.: Tests for global EDF schedulability analysis. Journal of Systems Architecture - Embedded Systems Design **57**(5), 487–497 (2011)
6. Bonifaci, V., Marchetti-Spaccamela, A.: Feasibility analysis of sporadic real-time multiprocessor task systems. In: M. de Berg, U. Meyer (eds.) ESA (2), *Lecture Notes in Computer Science*, vol. 6347, pp. 230–241. Springer (2010)
7. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
8. Cassez, F.: Timed games for computing wcet for pipelined processors with caches. In: 11th Int. Conf. on Application of Concurrency to System Design (ACSD'2011), pp. 195–204. IEEE Computer Society (2011)
9. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: T. Ball, R.B. Jones (eds.) CAV, *Lecture Notes in Computer Science*, vol. 4144, pp. 17–30. Springer (2006)
10. Delzanno, G., Raskin, J.F., Van Begin, L.: Covering sharing trees: a compact data structure for parameterized verification. International Journal on Software Tools for Technology Transfer (STTT) **5**(2–3), 268–297 (2004)
11. Doyen, L., Raskin, J.F.: Antichain algorithms for finite automata. In: J. Esparza, R. Majumdar (eds.) TACAS, *Lecture Notes in Computer Science*, vol. 6015, pp. 2–22. Springer (2010)



(a) Constrained deadlines ($0 < D_i \leqslant T_i$).



(b) Arbitrary deadlines ($0 < D_i \leqslant 4T_i$).

Fig. 17 Boxplots of automaton size vs. number of tasks with $T_{\max} = 6$. For each number of tasks, 200 task sets were generated and explored exhaustively.

12. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Inf. Comput. **205**(8), 1149–1172 (2007)
13. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: CAV, *Lecture Notes in Computer Science*, vol. 5643, pp. 263–277. Springer (2009)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
15. Goossens, J., Funk, S.: A note on task-parallelism upon multiprocessors. In: N. Fisher, R. Davis (eds.) RTSOPS 2010: 1st International Real-Time Scheduling Open Problems Seminar, pp. 18–19 (2010)
16. Goossens, J., Funk, S., Baruah, S.K.: EDF scheduling on multiprocessor platforms: some (perhaps) counterintuitive observations. In: Proceedings of the eighth International Conference on Real-time Computing Systems and Applications (RTCSA), pp. 321–330. Tokyo Japan (2002)
17. Goossens, J., Funk, S., Baruah, S.K.: Priority-driven scheduling of periodic task systems on multiprocessors. Real-Time Systems **25**(2-3), 187–205 (2003)
18. Guan, N., Gu, Z., Deng, Q., Gao, S., Yu, G.: Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In: R. Obermaisser, Y. Nah, P.P. Puschner, F.J. Rammig (eds.) SEUS, *Lecture Notes in Computer Science*, vol. 4761, pp. 263–272. Springer (2007)
19. Guan, N., Gu, Z., Lv, M., Deng, Q., Yu, G.: Schedulability analysis of global fixed-priority or EDF multiprocessor scheduling with symbolic model-checking. In: ISORC, pp. 556–560. IEEE Computer Society (2008)
20. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: FOCS, pp. 453–462 (1995)
21. Lindström, M., Geeraerts, G., Goossens, J.: A faster exact multiprocessor schedulability test for sporadic tasks. In: Proceedings of the 19th International Conference on Real-Time and Network Systems (RTNS 2011), September 29–30, Nantes, France, pp. 25–34 (2011)
22. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20**(1), 46–61 (1973)
23. Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential space. In: Proceedings of the 13th Annual Symposium on Switching and Automata Theory (swat 1972), pp. 125–129. IEEE Computer Society, Washington, DC, USA (1972). DOI 10.1109/SWAT.1972.29. URL http://dl.acm.org/citation.cfm?id=1437899.1438639
24. Sipser, M.: Introduction to the Theory of Computation, 1st edn. International Thomson Publishing (1996)
25. Tovar, E., Tsigas, P., Fouchal, H. (eds.): Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17–20, 2007. Proceedings, *Lecture Notes in Computer Science*, vol. 4878. Springer (2007)