Safe and Optimal Scheduling for Hard and Soft Tasks

Gilles Geeraerts
Université libre de Bruxelles, Brussels, Belgium
gigeerae@ulb.ac.be

Shibashis Guha
Université libre de Bruxelles, Brussels, Belgium
shibashis.guha@ulb.ac.be

Jean-François Raskin
Université libre de Bruxelles, Brussels, Belgium
jraskin@ulb.ac.be

Abstract

We consider a stochastic scheduling problem with both hard and soft tasks on a single machine. Each task is described by a discrete probability distribution over possible execution times, and possible inter-arrival times of the job, and a fixed deadline. Soft tasks also carry a penalty cost to be paid when they miss a deadline. We ask to compute an *online* and *non-clairvoyant* scheduler (i.e. one that must take decisions without knowing the future evolution of the system) that is *safe* and *efficient*. Safety imposes that deadline of hard tasks are never violated while efficient means that we want to minimise the mean cost of missing deadlines by soft tasks.

First, we show that the dynamics of such a system can be modelled as a finite Markov Decision Process (MDP). Second, we show that our scheduling problem is PP-hard and in EXPTime. Third, we report on a prototype tool that solves our scheduling problem by relying on the Storm tool to analyse the corresponding MDP. We show how antichain techniques can be used as a potential heuristic.

2012 ACM Subject Classification Theory of computation → Probabilistic computation, Computer systems organization → Real-time system specification, Computer systems organization → Embedded systems

Keywords and phrases Non-clairvoyant scheduling, hard and soft tasks, automatic synthesis, Markov decision processes

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2018.36

Acknowledgements This work was supported by the ARC project "Non-Zero Sum Game Graphs: Applications to Reactive Synthesis and Beyond" (Fédération Wallonie-Bruxelles).

1 Introduction

In modern real-time systems, we usually need to distinguish between two types of tasks: *hard tasks* that ought to be scheduled so that they meet their deadline with *absolute certainty* and *soft tasks* for which missing a deadline can be tolerated. Typically, hard tasks are vital for the correct execution of the system and missing a deadline for such tasks may have catastrophic consequences while missing a deadline of a soft task only degrades the overall performances of the system (as in a video decoding system, for example, where missing a deadline means skipping some video frames). An example of a system with both hard and soft tasks may consist of the computer system of a commercial aircraft, that must, at the same time, run

the avionics control, and the on-board entertainment system. Clearly, avionics controls are vital and most of these tasks will be *hard*, while missing a few frames in the video stream of the entertainment system only degrades the quality of the video; hence those tasks are *soft*. It is also usual to distinguish between tasks for which the inter-arrival time is fixed (such tasks are often called *periodic* tasks); and tasks for which the inter-arrival time is subject to uncertainty and specified by an interval constraint (such task are often called *sporadic* tasks). Most real systems naturally contain both periodic and sporadic tasks.

In this paper, we consider a rich formal model of infinite duration scheduling on a *single processor* that is applicable to systems with both periodic/aperiodic and hard/soft tasks. The tasks can be *preempted*. Additionally, we assume our schedulers to be *non-clairvoyant* in the sense that the execution and inter-arrival times of tasks are not known in advance and subject to uncertainty modelled by stochastic distributions. More precisely, each hard and soft task is characterised by a fixed deadline, and two discrete finite support distributions specifying its possible inter-arrival times and durations respectively. When a job associated to a task (i.e. a new instance of the task) arrives in the system, its execution time and the arrival time of the next job are not known but only the probability distribution over the possible execution times and arrival times are known. In addition, each soft task comes with a cost that is incurred each time a job of this task misses a deadline. The objective of a scheduler in this model is two-fold:

(i) the deadline of all jobs corresponding to hard tasks must be met with certainty; and

(ii) the expected mean cost of missing deadlines of jobs associated to soft tasks must be minimised.

Contributions. We define formally our scheduling problem as a non-standard optimisation problem on an MDP. That is, we consider MDPs with two *simultaneous* objectives: a *safety* objective asking that the deadline of each job associated to a hard task is met; and an *optimisation* objective asking to minimise the expected mean-cost of missing job deadlines associated to soft tasks. Second, we provide a worst-case exponential time algorithm (see Theorem 4) that decides the existence of a safe and optimal schedule, and provide a PP-hard lower bound[1], which is an improvement on the NP-Complete and coNP-Complete lower bounds that can be deduced from the literature (see related works below). Third, we propose a heuristic based on antichain techniques [16]: we identify a naturally occurring ordering on the states of the MDP, that can be used to prune the state space while computing the set of *safe* states. Thanks to the antichain technique, this set of safe states can also be described compactly. Finally, we have implemented a prototype tool for computing safe schedules on top of the probabilistic model-checker Storm [13], which we use to solve the *optimisation* part of our problem. We rely on the classical attractor algorithm to compute the set of states of the MDP from which the safety objective can always be satisfied. We also have implemented the antichain-based heuristic, and our experiments are encouraging: using the compact description of the safe states, we manage to produce a much smaller input file for Storm. Our algorithm works well for a small number of tasks and each task can have infinitely many jobs. Further an optimal schedule can be implemented simply as a *table lookup* and during runtime it requires minimal computation.

[1] Recall that PP is the class of problems that can be solved by a *probabilistic* Turing machine that operates in polynomial time [21].

To the best of our knowledge, there does not exist any scheduling algorithm in the literature that considers a cost model as ours as well as stochastic behaviour. In the appendix, we show that adapting the classical EDF scheduling policy to our setting yields scheduler that can be arbitrarily worse when compared to our optimal algorithm in terms of the expected mean-costs.

Related work. The schedulability of (hard) periodic tasks is a classical problem that has been studied in details in the literature, see e.g. [29, 26, 27, 14, 8]; and which has been shown to be coNP-complete in [26, 8] (see also [34, 25, 24, 7] where the tasks are not strictly periodic). It can be seen as a special case of our problem, where there are only hard tasks.

The *clairvoyant* scheduling of soft tasks (only) is also a classical problem that has attracted ample attention in the scheduling literature, see e.g. [41, 28, 6]. In [30], the authors consider a setting in which all the tasks have a mandatory (hard) part and an optional (soft) part that incurs a penalty when not executed; and show that cost minimisation is NP-complete when the optional tasks have arbitrary processing times. Again, this setting is a particular case of ours.

Finally, there are works in the literature that consider scheduling problems with both hard and soft tasks, see e.g. [10, 40, 11, 1]. A prominent line of works among them is based on the notion of *servers* [10, 40] to handle soft tasks. Algorithms for preemptive uniprocessor scheduling following this approach include Priority Exchange [40, 25], Sporadic Server [40, 38], Total Bandwidth Server [40], Earliest Deadline Late (EDL) Server [11, 39, 40], Constant Bandwidth Server [1], etc. However, those algorithms do not take into account a stochastic model of the tasks as in our problem nor a notion of deadline and cost for the soft tasks. The algorithm EDL is known to be optimal for dynamic priority assignment [11]. In Appendix B, we show that a version of EDL adapted to our setting can be arbitrarily worse in terms of the expected mean-cost when compared to our optimal algorithm.

The non-standard optimisation problem that we consider on MDP and which simultaneously asks for satisfying a safety and an expected mean-cost constraint is related to a recent line of works that mixes two-player zero sum games and MDPs, see e.g. [9, 3, 12].

2 Preliminaries

We denote by $\mathbb{N}$ the set of non-negative integer numbers, and by $\mathbb{Q}$ the set of rational numbers. For $n \in \mathbb{N}$, we use $[n]$ to denote $\{1, \ldots, n\}$ and $[n]_0$ to denote $\{0, 1, \ldots, n\}$. Given a finite set A, a (rational) *probability distribution* over A is a function $p\colon A \to [0,1] \cap \mathbb{Q}$ such that $\sum_{a \in A} p(a) = 1$. We denote the set of probability distributions on A by $\mathcal{D}(A)$. The *support* of the probability distribution p on A is $\mathsf{Supp}(p) = \{a \in A \mid p(a) > 0\}$. A distribution is called *Dirac* if $|\mathsf{Supp}(p)| = 1$.

Job Scheduling for both soft and hard tasks. We consider a system of n preemptive tasks $\{\tau_1, \ldots, \tau_n\}$ to be scheduled on a *single processor*. We identify all tasks τ_i with their unique respective index i. We assume that the time is discrete and measured in CPU ticks. Each task τ_i generates an infinite number of instances $\tau_{i,j}$, that we call *jobs*, with $j = 1, 2, \ldots$. We assume that all tasks are either *hard* or *soft* and denote by F and H the set of indexes of soft and hard tasks respectively (i.e. $i \in F$ iff τ_i is a soft task). Jobs generated by both hard and soft tasks are equipped with deadlines, which are relative to the respective arrival times of the jobs in the system. Jobs generated by hard tasks must complete before their

respective deadlines, but this is not mandatory for jobs generated by soft tasks. We also assume that tasks are independent, i.e. the scheduling a job of one task does not depend on another job belonging to some other task.

In order to make our model more realistic, we rely on a probabilistic model for the computation times of the jobs and on the time between the arrival of two successive jobs of the same task. Formally, for all $i \in [n]$, task τ_i is defined as a tuple $\langle I_i, \mathcal{C}_i, D_i, \mathcal{A}_i \rangle$, where:

- **(i)** $I_i \in \mathbb{N}$ is the arrival time of the first job of τ_i;
- **(ii)** $\mathcal{C}_i$ is a discrete probability distribution on the (finitely many) possible computation times of the jobs generated by τ_i;
- **(iii)** $D_i \in \mathbb{N}$ is the deadline of all jobs generated by τ_i which is relative to the arrival time of the jobs; and
- **(iv)** $\mathcal{A}_i$ is a discrete probability distribution on the (finitely many) possible inter-arrival times of the jobs generated by τ_i.

We note that $\max(\mathsf{Supp}(\mathcal{C}_i)) \leq D_i$, and throughout the paper, we assume that $D_i \leq \min(\mathsf{Supp}(\mathcal{A}_i))$ for all $i \in [n]$. In addition, we model the potential degradation in the quality when a soft task misses its deadline by a cost function $cost : F \to \mathbb{Q}_{\geq 0}$, that associates, to each soft task τ_j, a cost $c(j)$ which is incurred every time a job of τ_j misses its deadline.

One of the main contributions of this paper is to provide a formal model for such a system (see Section 3). We provide an intuitive explanation for now: each task τ_i releases a first job $\tau_{i,1}$ at time I_i. This job, like all other jobs of τ_i will request a CPU time which is chosen randomly according to $\mathcal{C}_i$. The deadline of $\tau_{i,1}$ is at time $I_i + D_i$. The next job $\tau_{i,2}$ will be released by τ_i at a time $I_i + \delta_2$, where δ_2 is chosen randomly according to $\mathcal{A}_i$, and so forth.

▶ **Example 1.** Consider a system with one hard task $\tau_h = \langle 0, \mathcal{C}_h, 2, \mathcal{A}_h \rangle$ s.t. $\mathcal{C}_h(1) = 1$ and $\mathcal{A}_h(3) = 1$; one soft task $\tau_s = \langle 0, \mathcal{C}_s, 2, \mathcal{A}_s \rangle$ s.t. $\mathcal{C}_s(1) = 0.4$, $\mathcal{C}_s(2) = 0.6$, and $\mathcal{A}_s(3) = 1$; and the cost function c s.t. $c(\tau_s) = 10$. This means that both tasks will submit their first job at time 0, both with deadlines at time $0 + 2 = 2$. Then, $\tau_{h,1}$ will have a computation time of 1, while $\tau_{s,1}$ will have a computation time which is either 1 (with probability 0.4) or 2 (with probability 0.6). Both tasks will submit new jobs $\tau_{h,2}$ and $\tau_{s,2}$ at time $0 + 3 = 3$. Each time a job of τ_s misses its deadline, a cost of 10 will be incurred.

Our goal is to find a *scheduler*, i.e. a function that, given the current state of the system, returns the identifier of the task that needs to be granted CPU access and ensuring that:

- **(i)** no job of the hard tasks misses its respective deadline; and
- **(ii)** the expected mean-cost incurred by the soft tasks missing their deadlines is minimised.

In Section 3, we model the problem as a game between two players: the *Scheduler* whose objectives are sketched above, and *TaskGen*, the task generator that generates jobs according to the semantics of the tasks, and whose goal is *antagonistic* to the scheduler's. Then, computing a scheduler will amount to computing a winning strategy of the *Scheduler* player. We now introduce the necessary notions to model this game with stochastic features.

Labelled Directed Graphs. A *labelled directed graph* (or graph for short) is a tuple $\mathcal{G} = \langle V, E, L \rangle$ where:

- **(i)** V is the finite set of vertices;
- **(ii)** $E \subseteq V \times V$ is the set of directed edges (sometimes called *transitions*); and
- **(iii)** $L : E \to A$ is the function labelling the edges by elements from some set A.

For a transition $e = (v, v')$, v is its *source*, denoted $\mathsf{src}(e)$, and v' its *destination* denoted $\mathsf{trg}(e)$.

Given $v \in V$, let $\mathsf{Succ}(v) = \{v' \in V \mid \exists (v, v') \in E\}$ be its set of successors, and $E(v) = \{e \mid \mathsf{src}(e) = v\}$ be its set of outgoing edges. We assume that for all $v \in V$: $\mathsf{Succ}(v) \neq \emptyset$, i.e. there is no deadlock. A *play* in a graph $\mathcal{G}$ from an initial vertex $v_{\mathsf{init}} \in V$ is an infinite sequence of transitions $\pi = e_0 e_1 e_2 \dots$ such that $\mathsf{src}(e_0) = v_{\mathsf{init}}$ and $\mathsf{trg}(e_i) = \mathsf{src}(e_{i+1})$ for all $i \geq 0$. The *prefix* up to the n-th vertex of π is the finite sequence $\pi(n) = e_0 e_1 \dots e_n$. We denote its last vertex by $\mathsf{Last}(\pi(n)) = \mathsf{trg}(e_n)$. The set of plays of $\mathcal{G}$ is denoted by $\mathsf{Plays}(\mathcal{G})$ and the corresponding set of prefixes is denoted by $\mathsf{Prefs}(\mathcal{G})$.

Weighted Markov Chains. A finite weighted *Markov chain* (MC, for short) is a tuple $M = \langle \mathcal{G}, Prob \rangle$, where $\mathcal{G} = \langle V, E, L \rangle$ is a graph with $L : E \mapsto \mathbb{Q}$ (i.e. edges are labelled by rational numbers that we call the *costs* of the edges), and $Prob : V \to \mathcal{D}(E)$ is a function that assigns a probability distribution on the set $E(v)$ of outgoing edges to all vertices $v \in V$. Given an initial vertex $v_{\mathsf{init}} \in V$, we define the set of possible *outcomes* in M as $\mathsf{Outs}_M(v_{\mathsf{init}}) = \{\pi = e_0 e_1 e_2 \dots \in \mathsf{Plays}(\mathcal{G}) \mid \mathsf{src}(e_0) = v_{\mathsf{init}} \wedge (\forall\, n \in \mathbb{N},\, e_{n+1} \in \mathsf{Supp}(\mathsf{trg}(e_n)))\}$. Let $V_{\mathsf{Outs}_M(v_{\mathsf{init}})} \subseteq V$ denote the set of vertices visited in the set of possible outcomes $\mathsf{Outs}_M(v_{\mathsf{init}})$. Finally, let us assume some measurable function $f : \mathsf{Plays}(\mathcal{G}) \to \mathbb{R}_{\geq 0}$ associating a rational value to each play of the MC. Since the set of plays of M forms a probability space, f is a random variable, and we denote by $\mathbb{E}^M_{v_{\mathsf{init}}}(f)$ the *expected value* of f over the set of plays starting from v_{init}.

Markov decision processes. A finite *Markov decision process* (MDP, for short) is a tuple $\Gamma = \langle V, E, L, (V_\Box, V_\bigcirc), A, Prob \rangle$, where:

- **(i)** A is a finite set of actions;
- **(ii)** $\langle V, E, L \rangle$ is a graph;
- **(iii)** the set of vertices V is partitioned into $V_\Box$ and $V_\bigcirc$;
- **(iv)** the graph is bipartite i.e. $E \subseteq (V_\Box \times V_\bigcirc) \cup (V_\bigcirc \times V_\Box)$, and the labeling function is s.t. $L(v, v') \in A$ if $v \in V_\Box$, and $L(v, v') \in \mathbb{Q}$ if $v \in V_\bigcirc$; and
- **(v)** $Prob$ assigns to each vertex $v \in V_\bigcirc$ a rational probability distribution on $E(v)$.

For all edges e, we let $\mathsf{cost}(e) = L(e)$ if $L(e) \in \mathbb{Q}$, and $\mathsf{cost}(e) = 0$ otherwise. We further assume that, for all $v \in V_\Box$, for all e, e' in $E(v)$: $L(e) = L(e')$ implies $e = e'$, i.e. an action identifies uniquely and outgoing edge.

An MDP can be interpreted as a game between two players: $\Box$ and $\bigcirc$ (Scheduler and TaskGen respectively), who own the vertices in $V_\Box$ and $V_\bigcirc$ respectively. A play in an MDP is a play in its underlying graph $\langle V, E, A \cup \mathbb{Q} \rangle$. We say that a prefix $\pi(n)$ of a play π belongs to player $i \in \{\Box, \bigcirc\}$, iff $\mathsf{Last}(\pi(n)) \in V_i$. The set of prefixes that belong to player i is denoted by $\mathsf{Prefs}_i(G)$. A play in the MDP is then obtained by the interaction of the two players as follows: if the current play prefix $\pi(n)$ belongs to $\Box$, she plays by picking an edge $e \in E(\mathsf{Last}(\pi(n)))$ (or, equivalently, an action that labels a necessarily unique edge from $\mathsf{Last}(\pi(n))$). Otherwise, when $\pi(n)$ belongs to $\bigcirc$, the next edge $e \in E(\mathsf{Last}(\pi(n)))$ is chosen randomly according to $Prob(\mathsf{Last}(\pi(n)))$. In both cases, the plays prefix is extended by e and the game goes *ad infinitum*.

A *strategy* of $\Box$ is a function $\sigma_\Box : \mathsf{Prefs}_\Box(G) \to E$, such that $\sigma_\Box(\rho) \in E(\mathsf{Last}(\rho))$ for all prefixes. A strategy $\sigma_\Box$ is *memoryless* if for all finite prefixes ρ_1 and $\rho_2 \in \mathsf{Prefs}(\mathcal{G})$: $\mathsf{Last}(\rho_1) = \mathsf{Last}(\rho_2)$ implies $\sigma_\Box(\rho_1) = \sigma_\Box(\rho_2)$. From now on, we will consider mainly memoryless strategies. Let $\Gamma = \langle V, E, L, (V_\Box, V_\bigcirc), A, Prob \rangle$ be an MDP and let $\sigma_\Box$ be a *memoryless* strategy. Then, assuming that $\Box$ plays according to $\sigma_\Box$, we can express the behaviour of Γ as an MC $\Gamma[\sigma_\Box]$, where the probability distributions reflect the stochastic choices of $\bigcirc$. Formally, $\Gamma[\sigma_\Box] = \langle V_\bigcirc, E', L', Prob' \rangle$, where $(v, v') \in E'$ iff there is $\hat{v}$ s.t.:

(i) $(v, \hat{v}) \in E$;
(ii) $\sigma_\Box(\hat{v}) = v'$; and
(iii) $Prob(\hat{v}, v') = Prob'(v, v')$.
Further, for all $e \in E'$, we have $L'(e) = L(e)$.

Safety synthesis. Given an MDP $\Gamma = \langle V, E, L, (V_\Box, V_\bigcirc), A, Prob\rangle$, an initial vertex $v_{\mathsf{init}} \in V$, and a set $V_{\mathsf{safe}} \subseteq V$ of so-called *safe vertices*, the *safety synthesis problem* is to decide whether $\Box$ has a strategy $\sigma_\Box$ such that $V_{\mathsf{Outs}_{\Gamma[\sigma_\Box]}(v_{\mathsf{init}})} \subseteq V_{\mathsf{safe}}$, that is, all the plays obtained when $\Box$ plays according to $\sigma_\Box$ visit only the safe vertices. The safety synthesis problem is decidable in polynomial time for MDPs. Indeed, since probabilities do not matter for this problem, the MDP can be regarded as a plain two-player game played on graphs (like in [42]), and the classical *attractor* algorithm can be used (see Appendix A).

Expected mean cost threshold synthesis. Let us first associate a value, called the *mean cost* $\mathsf{MC}(\pi)$ to all plays π in an MDP $\Gamma = \langle V, E, L, (V_\Box, V_\bigcirc), A, Prob\rangle$. First, for a prefix $\rho = e_0 e_1 \dots e_{n-1}$, we define $\mathsf{MC}(\rho) = \frac{1}{n}\sum_{i=0}^{i=n-1} \mathsf{cost}(e_i)$ (recall that $\mathsf{cost}(e) = 0$ when $L(e)$ is an action). Then, for a play $\pi = e_0 e_1 \dots$, we have $\mathsf{MC}(\pi) = \limsup_{n\to\infty} \mathsf{MC}(\pi(n))$. Observe that MC is a measurable function. Then, the *expected mean-payoff threshold synthesis* problem is to decide whether $\Box$ has a strategy $\sigma_\Box$ such that $\mathbb{E}^{\Gamma[\sigma_\Box]}_{v_{\mathsf{init}}}(\mathsf{MC}) \leq \lambda$ for some initial vertex $v_{\mathsf{init}} \in V$ and threshold $\lambda \in \mathbb{Q}$. Such strategies are called *optimal*, and it is well-known that, if such an optimal strategy exists, then, there is one which is *memoryless*. Moreover, this problem can be solved in polynomial time through linear programming [17] or in practice using value iteration (as implemented, for example, in the tool Storm [13]).

3 Modelling the system as an MDP

Let us fix a system of tasks $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$ and a cost function *cost* and let us model our scheduling problem by means of an MDP Γ_τ. This will provide us with a precise and formal definition of the problem, and will allow us to rely on automatic tools (such as Storm [13], see Section 6) to solve it. In order to define Γ_τ, it is easier to first build an *infinite* MDP $\Gamma = \langle V, E, L, (V_\Box, V_\bigcirc), \ell, Prob\rangle$. Then, Γ_τ will be the (finite) portion of Γ that is reachable from some designated initial state v_{init}. In our model, $\Box$ models the *Scheduler* and $\bigcirc$ models the task generator (abbreviated *TaskGen*).

Modelling the system states. Since $D_i \leq \min(\mathsf{Supp}(\mathcal{A}_i))$ for all tasks τ_i, there can be at most one job of each task at a time t that can be scheduled at t or later. Thus when the system executes, we keep information related to only one job per task. For each task τ_i, at every time, in the vertices of the MDP we maintain the following information about the current job in the system at that time:
(i) a *distribution* c_i over the job's possible remaining computation times (rct);
(ii) the time d_i up to its deadline; and
(iii) a distribution a_i over the possible times up to the next arrival of a new job of τ_i.
We also have a special vertex $\bot$ that will be reached when a hard task misses a deadline. Formally: $V_\Box = \big(\mathcal{D}([C_{\max}]_0) \times [D_{\max}]_0 \times \mathcal{D}([A_{\max}]_0)\big)^n \times \{\Box\} \cup \{\bot\}$ and $V_\bigcirc = \big(\mathcal{D}([C_{\max}]_0) \times [D_{\max}]_0 \times \mathcal{D}([A_{\max}]_0)\big)^n \times \{\bigcirc\}$; where $C_{\max} = \max_i(\max(\mathsf{Supp}(\mathcal{C}_i)))$, $D_{\max} = \max_i(\{D_i\})$ and $A_{\max} = \max_i(\max(\mathsf{Supp}(\mathcal{A}_i)))$. For a vertex $v = \big((c_1, d_1, a_1), \dots (c_n, d_n, a_n)\big)$, we let $\mathsf{active}(v) = \{i \mid c_i(0) \neq 1 \text{ and } d_i > 0\}$ and $\mathsf{dlmiss}(v) = \{i \mid c_i(0) = 0 \text{ and } d_i = 0\}$. Intuitively, $\mathsf{active}(v)$ is the set of tasks that have an active job in v, that is one which has not finished

and whose deadline has not passed yet; and $\mathsf{dlmiss}(v)$ is the set of tasks that have missed a deadline *for sure* in v (observe that for $c_i(0) > 0$, the task *could* complete now and does not miss a deadline for sure). In v, for every task $i \in [n]$, the tuple (c_i, d_i, a_i) is called its *configuration*.

Distribution updates. Let us now introduce the dec and norm functions that will be useful when we will need to update the knowledge of the Scheduler. For example, consider a state where $c_i(1) = 0.5$, $c_i(4) = 0.1$ and $c_i(5) = 0.4$ for some i, and where τ_i is granted one CPU time unit. Then, all elements in the support of c_i should be decremented, yielding c'_i with $c'_i(0) = 0.5$, $c'_i(3) = 0.1$ and $c'_i(4) = 0.4$. Since $0 \in \mathsf{Supp}(c'_i)$, the current job of τ_i *could* now terminate with probability $c'_i(0) = 0.5$, or continue running, which will be observed by the Scheduler player. In the case where the job does not terminate, the probability mass must be redistributed to update Scheduler's knowledge, yielding the distribution c''_i with $c''_i(3) = 0.2$ and $c''_i(4) = 0.8$. Formally, let p and p' be probability distributions on $\mathbb{N}$ s.t. $0 \notin \mathsf{Supp}(p)$. Then, we let $\mathsf{dec}(p)$ and $\mathsf{norm}(p')$ be probability distributions on $\{x - 1 \mid x \in \mathsf{Supp}(p)\}$ and $\mathsf{Supp}(p') \setminus \{0\}$ respectively s.t.:

$$\text{for all } x \in \mathsf{Supp}(p) : \mathsf{dec}(p)(x-1) = p(x)$$
$$\text{for all } x \in \mathsf{Supp}(p') \setminus \{0\} : \mathsf{norm}(p')(x) = \frac{p'(x)}{\sum_{x \geq 1} p'(x)}.$$

Observe that, when $0 \notin \mathsf{Supp}(p')$, then $\mathsf{norm}(p') = p'$.

Possible moves of the Scheduler. The possible actions of Scheduler are to schedule an active task or to idle the CPU. We model this by having, from all states $v \in V_\Box$ one transition labelled by some element from $\mathsf{active}(v)$, or by ε (no job gets scheduled). The effect of the transition models the elapsing of one clock tick.

Formally, fix $v = \big((c_1, d_1, a_1), \ldots, (c_n, d_n, a_n), \Box\big) \in V_\Box$ s.t. $0 \notin \mathsf{Supp}(c_i)$ and $0 \notin \mathsf{Supp}(a_i)$ for all $i \in [n]$. Then, there is $e = (v, v') \in E$ with $L(e) \in [n] \cup \{\varepsilon\}$ and $v' = \big((c'_1, d'_1, a'_1), \ldots, (c'_n, d'_n, a'_n), \bigcirc\big)$ iff the following four conditions hold:

- **(i)** $L(e) = i \in [n]$ implies that $i \in \mathsf{active}(v)$ and $c'_i = \mathsf{dec}(c_i)$, i.e. if a task is scheduled, it must be active, and its rct is decremented;
- **(ii)** for all $j \in [n] \setminus \{L(e)\}$: $c'_j = c_j$, i.e. the rct of all the other tasks does not change;
- **(iii)** for all $j \in [n]$: $d'_j = \max(d_j - 1, 0)$, i.e. the deadline is one time unit closer, if not reached yet;
- **(iv)** for all $j \in [n]$: $a'_j = \mathsf{dec}(a_j)$, i.e. we decrement the time to next arrival of all tasks.

Observe that when a *soft* task misses a deadline, we maintain the positive rct of the job in the next state: this will be used as a marker to ensure that the associated cost is paid when a new job of the same task will arrive. For example, consider a state (with one soft task) $\big((c, 0, a), \Box\big)$ with $c(2) = 1$ and $a(1) = 1$ i.e. the task has reached its deadline but still need 2 time units to complete and the next job arrives in 1 time unit. Then, the successor state will be $\big((c, 0, a'), \bigcirc\big)$ with $a'(0) = 1$, from which a new job will be submitted, which will incur a cost (see action $killANDsub$ in the next paragraph).

Possible moves of the Task Generator. The moves of TaskGen (modelled by $\bigcirc$) consist in selecting, for each task one possible *action* out of four: either nothing (ε); or

- **(i)** to finish the current job without submitting a new one (fin); or
- **(ii)** to submit a new job while the previous one is already finished (sub); or
- **(iii)** to submit a new job and kill the previous one, in the case of a soft task ($killANDsub$), which will incur a cost.

Formally, let $Actions = \{fin, sub, killANDsub, \varepsilon\}$. To define Γ_τ, we introduce a function $\mathcal{L} : (V_\bigcirc \times V_\Box) \mapsto Actions^n$, i.e. $\mathcal{L}(e,i)$ is the action of $\bigcirc$ corresponding to τ_i on edge e. Fix a state $v = \big((c_1,d_1,a_1),\ldots,(c_n,d_n,a_n),\bigcirc\big) \in V_\bigcirc$. Let $\hat{v} = \big((\hat{c_1},\hat{d_1},\hat{a_1}),\ldots,(\hat{c_n},\hat{d_n},\hat{a_n}),\Box\big) \in V_\Box$ be such that that $\hat{v} \xrightarrow{i} v$. Note that there is a unique such $\hat{v}$ from which action i can be done to reach v. We consider two cases. Either $\mathsf{dlmiss}(v) \cap H \neq 0$, i.e., a hard task has just missed a deadline. In this case, the only transition from v is $e = (v, \bot)$ with $\mathcal{L}(e,i) = \varepsilon$ for all $i \in [n]$. Otherwise, there is an edge $e = (v,v')$ with $v' = \big((c'_1,d'_1,a'_1),\ldots,(c'_n,d'_n,a'_n),\Box\big)$ iff for all $i \in [n]$, one of the following holds:

1. $\mathcal{L}(e,i) = fin$, $\min(\mathsf{Supp}(\hat{c}_i = 1))$, $a_i(0) \neq 1$, $c'_i(0) = 1$, $d'_i = d_i$, and $a'_i = \mathsf{norm}(a_i)$. The current job of τ_i finishes now ($c_i(0) > 0$) and the next arrival will occur in the future ($a_i(0) \neq 1$), according to the probability distribution $\mathsf{norm}(a_i)$.
2. $\mathcal{L}(e,i) = sub$, $c_i(0) > 0$, $a_i(0) > 0$, $c'_i = \mathcal{C}_i$, $d'_i = D_i$, and $a'_i = \mathcal{A}_i$. In this case, we assume that the previous job of τ_i has completed ($c_i(0) > 0$) and we let τ_i submit a new job (see the new values c'_i, d'_i and a'_i); or
3. $\mathcal{L}(e,i) = killANDsub$, $c_i(0) \neq 1$, $a_i(0) > 0$, $c'_i = \mathcal{C}_i$, $d'_i = D_i$, and $a'_i = \mathcal{A}_i$. In this case, τ_i (necessarily a soft task) submits a new job, and kills the previous one (there is possibly some remaining rct as $c_i(0) \neq 1$); or
4. $\mathcal{L}(e,i) = \varepsilon$, $a_i(0) \neq 1$, either $c'_i = \mathsf{norm}(c_i)$ or $c'_i(0) = c_i(0) = \hat{c}_i(0) = 1$, $d'_i = d_i$ and $a'_i = \mathsf{norm}(a_i)$. No action is performed on τ_i which must not submit a new job now ($a_i(0) \neq 1$) and does not finish now. For $c'_i(0) = c_i(0) = \hat{c}_i(0) = 1$, it denotes that the job already finished during a previous clock tick. The knowledge of the scheduler (c' and a') is updated accordingly.

The cost of and edge e is: $L(e) = c = \sum_{i:L(e,i)=killANDsub} cost(i)$. As said earlier, the cost is incurred when the $killANDsub$ action is performed by some task τ_i, although the deadline miss might have occurred earlier. Finally, the probability of an edge e is $Prob(e) = \prod_{i\in[n]} p_i$, where, for all $i \in [n]$:

$$p_i = \begin{cases} c_i(0)\cdot(1-a_i(0)) & \text{if } \mathcal{L}(e,i) = fin \\ c_i(0)\cdot a_i(0) & \text{if } \mathcal{L}(e,i) = sub \\ (1-c_i(0))\cdot a_i(0) & \text{if } \mathcal{L}(e,i) = killANDsub \\ (1-c_i(0))\cdot(1-a_i(0)) & \text{if } \mathcal{L}(e,i) = \varepsilon \text{ and } c_i(0) \neq 1 \\ 1-a_i(0) & \text{if } \mathcal{L}(e,i) = \varepsilon \text{ and } c_i(0) = 1. \end{cases}$$

Finally, the initial vertex is $v_{\mathsf{init}} = \big((c_0,d_0,a_0),\ldots,(c_n,d_n,a_n),\Box\big) \in V_\Box$ s.t. for all $i \in [n]$: $(c_i,d_i,a_i) = (\mathcal{C}_i, D_i, \mathcal{A}_i)$ if $I_i = 0$; and $(c_i,d_i,a_i) = (c,0,a)$ with $c(0) = 1$ and $a(I_i) = 1$ otherwise. The finishes the definition of Γ. As explained above, the MDP Γ_τ modelling our problem is the portion of Γ that is reachable from v_{init}. One can check, by a careful inspection of the definitions, that Γ_τ is indeed finite. Let us now illustrate these definitions.

▶ **Example 2.** Figure 1 presents an excerpt of the MDP Γ_τ built from the set of tasks $\tau = \{\tau_h, \tau_s\}$ of Example 1. We denote a distribution p with support $\{x_1, x_2, \ldots, x_n\}$ by $[x_1, p(x_1); x_2, p(x_2); \ldots; x_n p(x_n)]$. When p is s.t. $p(x) = 1$ for some x, we simply denote p by x. States from $V_\Box$ and $V_\bigcirc$ are depicted by rectangles and rounded rectangles respectively. Each state is labelled by (c_h, d_h, a_h) on the top and (c_s, d_s, a_s) below.

A strategy to avoid missing a deadline of τ_h consists in first scheduling τ_s, then τ_h. One then reaches the left-hand part of the graph from which $\Box$ can avoid $\bot$ irrespective of whatever $\bigcirc$ does. Note that other safe strategies are possible: the first step of our algorithm

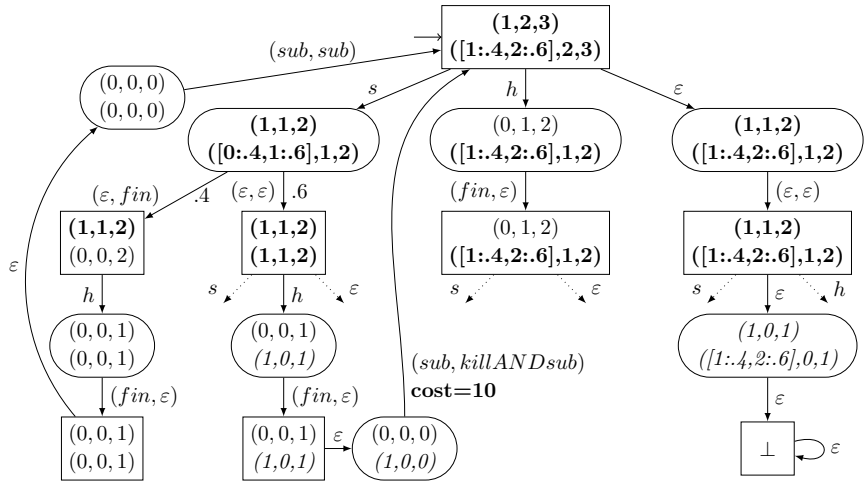

Figure 1 An excerpt of the MDP for Example 1. Tasks in **bold** are active, tasks in *italics* have missed a deadline.

is actually to compute all the *safe* nodes (i.e. those from which $\Box$ can ensure to avoid $\bot$), and then to look for an optimal one (wrt to the cost of missing deadlines for soft tasks) among those.

From state $\langle (1,1,2), ([0{:}.4,1{:}.6],1,2)\rangle$, $\bigcirc$ chooses whether τ_s finishes or not with respective probabilities 0.4 and 0.6. In the latter case, τ_s will miss its deadline, which incurs a cost on the edge where the $killANDsub$ action occurs.

Equipped with these definitions, we can define the problem that we want to solve. The **Safe and optimal scheduler synthesis** problem is stated as, given a set of real-time tasks τ partitioned into hard and soft tasks, and a rational threshold λ whether there exists, in the MDP Γ_τ a strategy $\sigma_\Box$ of $\Box$ s.t.:

- **(i)** $\bot \notin V_{\mathsf{Outs}_{\Gamma_\tau[\sigma]}(v^\tau_{\mathsf{init}})}$, i.e. no hard task misses its deadline. Strategies that enforce this objective are called *safe* strategies; and
- **(ii)** $\mathbb{E}^{\Gamma_\tau[\sigma]}_{v^\tau_{\mathsf{init}}}(\mathsf{MC}) \leq \lambda$, i.e. the expectation of the mean-cost (due to the deadline misses by the soft tasks) is at most λ.

This strategy $\sigma_\Box$ constitutes our scheduler. We will see in the next section that when such a strategy exists, then there exists one which is also *memoryless*.

▶ **Example 3.** Let us continue with Example 2. There are two optimal memoryless strategies, one in which the Scheduler first chooses to execute τ_h, then τ_s; and another where τ_s is scheduled for 1 time unit, and then preempted to let τ_h execute. Since the period of τ_s is 3 and the cost of missing a deadline is 10, for both of these optimal strategies, the soft task's deadline is missed with probability 0.6 during each period and hence the mean-cost is 2. Observe that there is another safe schedule that is not optimal is one in which only τ_h is granted CPU access, and τ_s is never scheduled thus giving a mean-cost of $\frac{10}{3}$.

4 Algorithm and Complexity

In this section, we show that the safe and optimal scheduler synthesis problem is PP-hard and in EXPTIME. We start with the upper bound first.

▶ **Theorem 4.** *The safe and optimal scheduler synthesis problem can be solved in* EXPTIME.

Proof. We sketch an exponential time algorithm that solves our problem. First, we build the MDP $\Gamma_\tau = \langle V, E, A, Prob\rangle$ according to the above definitions. Note that the supports of the distributions and the deadline given in binary, the size of this MDP is exponential in the size of the task set, hence Γ_τ can be built in exponential time. Then, we run the attractor algorithm on Γ_τ (see Appendix A), using $\{\bot\}$ as the set of *unsafe* vertices. This takes polynomial time in the size of Γ_τ, hence exponential time in the size of τ. We obtain a set $V_{unsafe} \subseteq V$ of *losing vertices*. That is, V_{unsafe} contains all the vertices from which $\Box$ cannot guarantee that all the hard tasks will never miss deadlines. We then prune Γ_τ by removing all the vertices of V_{unsafe}, and obtain $\Gamma' = \langle V \setminus V_{unsafe}, E', A, Prob\rangle$. Observe that, by definition of the attractor, whenever a vertex $v \in V_\bigcirc$ has a successor in V_{unsafe}, then $v \in V_{unsafe}$ too. So, the pruning operation either keeps a vertex $v \in V_\bigcirc$ with all its successors, or remove it. The corresponding edges are also removed. Hence, the $Prob$ function is still a probability distribution and Γ' is still an MDP. By property of the attractor, the possible strategies for $\Box$ in Γ' are exactly all the *safe* strategies in Γ_τ. Hence, we can now solve our problem by applying some classical polynomial time algorithm [36, 35] to solve the mean-cost threshold synthesis problem in Γ', and we have the guarantee that for all strategies $\sigma_\Box$: $\mathbb{E}^{\Gamma'[\sigma_\Box]}_{v^\tau_{\mathsf{init}}}(\mathsf{MC}) \leq \lambda$ iff $\bot \notin V_{\mathsf{Outs}_{\Gamma_\tau[\sigma_\Box]}(v^\tau_{\mathsf{init}})}$ and $\mathbb{E}^{\Gamma_\tau[\sigma_\Box]}_{v^\tau_{\mathsf{init}}}(\mathsf{MC}) \leq \lambda$.

Observe that those algorithms compute *memoryless* optimal strategies that map all vertices in $V_\Box$ to an action in $[n] \cup \{\varepsilon\}$, representing which task (if any) should be granted CPU access. This strategy is thus the actual scheduler, and we are done. ◀

Let us now turn our attention to a lower bound on the complexity. As explained in the *related works* section, our problem subsumes classical scheduling problems that are known to be coNP-complete, like the periodic (hard) task scheduling problem [26, 8], and NP-complete, like the clairvoyant scheduling problem for soft tasks [30]. The proof of task scheduling problem for sporadic hard tasks in [7] also applies to our case giving coNP-completeness in a system with only hard tasks that are neither periodic nor sporadic[2]. We now provide a stronger lower bound by establishing PP-hardnes. Recall that PP is the class of languages $L \subseteq \Sigma^*$ recognised by a probabilistic polynomial-time Turing machine M with access to a fair coin such that for all $w \in \Sigma^*$, we have $w \in L$ if and only if M accepts w with probability at least $\frac{1}{2}$. The class PP contains NP, is closed under complement [37] and hence also contains the class coNP. Further, the class PP is contained in PSPACE.

▶ **Theorem 5.** *The safe and optimal schedule synthesis problem is* PP*-hard.*

Proof. We show a reduction from k-th largest subset which has recently been shown to be PP-complete [22]. The k-th largest subset problem is stated as given a finite set A, a size function $h : A \to \mathbb{N}$ assigning strictly positive integer values to elements of A, and two naturals $K, L \in \mathbb{N}$, decide if there exist K or more distinct subsets $S_j \subseteq A$, where $1 \leq j \leq K$, such that $\sum_{o \in S_j} h(o) \leq L$ for all these K or more subsets.

Let $|A| = n$, and $B = \sum_{o \in A} h(o)$. Given an instance of k-th largest subset, we construct a system of n hard tasks and one soft task. Let H denote the set of hard tasks. The first instance of each hard task arrives at time 0. The computation time is $e_i = h(o_i)$ with probability $\frac{1}{2}$, and $e_i = 0$ with probability $\frac{1}{2}$, the deadline $d = B$, and the inter-arrival time $p_i = B$ for all $1 \leq i \leq n$.

The arrival time of the first instance of the soft task is L, its computation time is $B - L$, (relative) deadline is $B - L$ and inter-arrival time is B.

[2] Note that in sporadic tasks, only the minimum inter-arrival time for a task is specified.

Suppose there is a solution to the instance of k-th largest subset problem, i.e. there are at least $r \geq K$ subsets of A such that the sum of the elements in each S_j, where $j \in [r]$ is less than L. Hence in the system constructed above, there are $r \geq K$ subsets of the set H such that corresponding to each subset S_j, where $i \in [r]$, for each $i \in [n]$, hard task i executes with time $h(o_i)$ if $o_i \in S_j$, else i executes with time 0.

Note that since each hard task has two possible computation times that are 0 and $h(o_j)$, there are 2^n combinations of computation times for all the hard tasks, each such combination having a probability of $\frac{1}{2^n}$, and r of them finish before L and for each of these r combinations, the soft task executes to completion. Thus the cost is 0 for r of these combinations, that is, with probability $\frac{r}{2^n}$, while the cost is 1 with probability $1 - \frac{r}{2^n}$. Hence the expected cost is $1 - \frac{r}{2^n}$ over a time period B. The expected mean-cost is $\frac{1}{B} \cdot (1 - \frac{r}{2^n})$. So the expected mean-cost is indeed less than or equal to $\frac{1}{B} \cdot (1 - \frac{K}{2^n})$ iff there are at least K subsets of A each of whose elements sum up to L or less. ◀

As a consequence, our EXPTIME upper bound cannot be improved substantially unless P=PP. We note that our proof implies that we cannot have a short certificate indicating the absence of a safe schedule that meets some minimal performance for the soft tasks unless PP = coNP = NP.

Finally, we note that in our reduction, we associate a system with only one soft task to each instance of the K-th largest subset sum problem. But our reduction can be adapted by turning all *hard tasks* into *soft tasks* with costs > 1. In place of each hard task, we have a soft task, each parameter of the soft task remains the same as the hard task, and the cost of missing the deadline for the soft task is strictly greater than 1 while we have only one soft task as in the previous case whose cost is 1. Arguing as above, we see that minimum expected mean-cost is less than or equal to $\frac{1}{B} \cdot (1 - \frac{K}{2^n})$ iff there are at least K subsets of A each of whose elements sum up to L or less. Hence, we obtain the same lower bound in the case where we have only soft tasks:

▶ **Corollary 6.** *The safe and optimal scheduler synthesis problem is* PP*-hard even in a system with only soft tasks.*

This shows that the non-clairvoyant scheduling of soft tasks where the tasks are described using probability distributions is computationally more difficult than the clairvoyant version (unless NP = PP).

5 A symbolic data-structure for the safety game

Our last theoretical contribution is to propose an antichain-based [16] heuristic to mitigate the high complexity of the problem, and solve the *safety* part of the game in an efficient way (in practice). The core of this approach consists in identifying an ordering $\succeq \subseteq V_\Box \times V_\Box$ on the vertices of $\Box$ and that can be interpreted intuitively as follows: $v_1 \succeq v_2$ means that v_1 is *as difficult* as v_2 (from the point of view of $\Box$, i.e. the Scheduler player). Thus, if $\Box$ has a safe strategy from v_1, then she also has a safe strategy from v_2 (Theorem 9). This implies that the set of safe $\Box$ vertices, (that our algorithm computes in the first place) is *downward-closed* for $\succeq$, a special structure that we will exploit in our implementation (see Section 6).

Let $v^1 = \big((s_1^1, s_2^1, \ldots, s_n^1), \Box\big)$, and $v^2 = \big((s_1^2, s_2^2, \ldots, s_n^2), \Box\big)$ be two vertices of $\Box$ where $s_i^j = (c_i^j, d_i^j, a_i^j)$ for all $j \in \{1, 2\}$ and $i \in [n]$. Intuitively, in order to make sure that, for all tasks $i \in [n]$ its configuration s_i^1 is *at least as difficult as* s_i^2, we could request that:

(i) for all $e_i^2 \in \mathsf{Supp}(c_i^2)$, there is $e_i^1 \in \mathsf{Supp}(c_i^1)$ s.t.: $d_i^2 - e_i^2 \geq d_i^1 - e_i^1$; and
(ii) $d_i^1 \leq d_i^2$; and
(iii) $\mathsf{Supp}(a_i^1) \supseteq \mathsf{Supp}(a_i^2)$.

Indeed, condition (i) means that the amount of work needed for τ_i to complete in v^1 is at least the amount of work needed in v^2, whatever the choices of the task generator (observe that we ignore the probabilities here since we are only interested in safety). Condition (ii) says that the deadline is closer in v^1 than in s^2. Condition (iii) ensures that all next arrivals that can occur in v^2 can also occur in v^1, hence, all the actions that $\bigcirc$ will be able to play from the successors of v^2 should also be present from the successors of v^1.

Let us now argue that we can actually simplify these conditions. Assume the three conditions hold on v^1 and v^2, and consider (iii) i.e. $\mathsf{Supp}(a_i^1) \supseteq \mathsf{Supp}(a_i^2)$. Since the a_i parameters of all tasks are initialised to the same value $\mathcal{A}_i$, this implies that the job of τ_i in v^2 has been submitted earlier than the corresponding job in v^1, which implies that $d_i^1 \geq d_i^2$. This last inequality together with (ii) implies that $d_i^1 = d_i^2$, which means that the two jobs of τ_i in both states have actually been submitted at the same time, hence we must also have $a_i^1 = a_i^2$. This motivates our definition of the $\succeq$ order:

▶ **Definition 7.** Let $v^1 = \big((s_1^1, s_2^1, \ldots, s_n^1), \square\big)$, and $v^2 = \big((s_1^2, s_2^2, \ldots, s_n^2), \square\big)$ be two states of $\square$ where $s_i^j = (c_i^j, d_i^j, a_i^j)$ for all $j \in \{1, 2\}$ and $i \in [n]$. Then, $v^1 \succeq v^2$ iff: for all $i \in [n]$, there exists an injective function $f : \mathsf{Supp}(c_i^2) \to \mathsf{Supp}(c_i^1)$ s.t. for all $e_2 \in \mathsf{Supp}(c_i^2)$: $f(e_2) \geq e_2$, $d_i^1 = d_i^2$ and $a_i^1 = a_i^2$.

Observe that this ordering is defined on the *structure* of the states of the MDP, so it is easy to test simply by inspecting v^1 and v^2. In order to show that $\succeq$ has the right properties, we rely on a variation of the notion of alternating simulation [4], which we adapt to our setting. Fix an MDP $\gamma_\tau = \langle V, E, L, (V_\square, V_\bigcirc), \ell, Prob\rangle$. Then, a relation $\mathcal{R} \subseteq V_\square \times V_\square$ is an alternating simulation relation iff for all $(v_1, v_2) \in \mathcal{R}$, the following holds. For all $v_1' \in \mathsf{Succ}(v_1)$, there is $v_2' \in \mathsf{Succ}(v_2)$ s.t.:

(i) $L(v_2, v_2') \in \{L(v_1, v_1'), \varepsilon\}$; and
(ii) for all $v_2'' \in \mathsf{Succ}(v_2')$ there is $v_1'' \in \mathsf{Succ}(v_1')$ s.t. $\mathcal{L}(v_1', v_1'') = \mathcal{L}(v_2', v_2'')$ and $(v_1'', v_2'') \in \mathcal{R}$.

Then, we can show that:

▶ **Lemma 8.** *$\succeq$ is an alternating simulation relation.*

Proof. Let $(v_1, v_2) \in \succeq$ with $v_1 = (s_{1,1}, s_{2,1}, \ldots s_{n,1}, \square)$ and $v_1 = (s_{1,2}, s_{2,2}, \ldots s_{n,2}, \square)$ Recall that $v_1, v_2 \in V_\square$. Let $i \in [n] \cup \{\varepsilon\}$ be the action of $\square$ from v_1, let $v_1 \xrightarrow{i} v_1'$, and let $s_{i,1} = \langle c_1, d, a\rangle$ and $s_{i,2} = \langle c_2, d, a\rangle$. Since $v_1 \succeq v_2$, by definition of $\succeq$, there exists an injective function $f : \mathsf{Supp}(c_2) \to \mathsf{Supp}(c_1)$ as defined above. Let $e = \min(\mathsf{Supp}(c_2))$. Consider the action from v_2 as $v_2 \xrightarrow{\varepsilon} v_2'$ if $f(e) > e$, else $v_2 \xrightarrow{i} v_2'$.

Now since a_i is the same in both v_1' and v_2', we see that for every action b of $\bigcirc$ such that $v_2' \xrightarrow{b} v_2''$, we have a v_1'' such that $v_1' \xrightarrow{b} v_1''$, and $v_1'', v_2'' \in V_\square$. From the transitions of the MDP as defined in Section 3, it is not difficult to see that $(v_1'', v_2'') \in \succeq$, and we are done. ◀

From this Lemma, and from the fact that the objective of the *safety* part of the game is to avoid reaching $\bot$, we can now deduce that $\succeq$ has the desired property. More precisely, we show that, if $v_1 \succeq v_2$ and Scheduler can schedule all hard tasks from v_1, then we can find a so-called $\succeq$-strategy σ' that is also winning *and* that takes the same choices (when possible, that is, when the same tasks are active) from all pairs of states which are comparable (in particular, from s_2). Formally, we call a strategy $\sigma : V_\square \to [n] \cup \{\varepsilon\}$ *$\succeq$-compatible* iff for all v_1, v_2 with $v_1 \succeq v_2$: either $\sigma(v_1) = \sigma(v_2)$ or $\sigma(v_2) = \varepsilon$. That is, either σ schedules the same task from both states, or it idles the CPU in the "easier" state v_2.

▶ **Theorem 9.** *For all $\Box$ vertices v_1 and v_2 with $v_1 \succeq v_2$, if $\Box$ (Scheduler player) has a safe strategy from v_1, then she has a $\succeq$-compatible safe strategy σ from v_2 (also safe from v_1).*

Proof. Let $v_1 = \big((c_1^1, d_1^1, a_1^1), \ldots, (c_n^1, d_n^1, a_n^1), \Box\big)$ and $v_2 = \big((c_1^2, d_1^2, a_1^2), \ldots, (c_n^2, d_n^2, a_n^2), \Box\big)$ respectively. Given a winning strategy σ' from v_1, we construct an $\succeq$-compatible winning strategy σ from v_1 as follows. Consider $V_{\mathsf{Outs}_{\Gamma(\sigma')}(v_1)}$, the set of all vertices visited from v_1 when σ' is played from v_1. We define σ such that for all $\tilde{v}_1 \in V_{\mathsf{Outs}_{\Gamma(\sigma')}(v_1)}$, we have $\sigma(\tilde{v}_1) = \sigma'(\tilde{v}_1)$. For all vertices $\hat{v}_1 = \big((\hat{c}_1, \hat{d}_1, \hat{a}_1), (\hat{c}_2, \hat{d}_2, \hat{a}_2), \ldots, (\hat{c}_n, \hat{d}_n, \hat{a}_n), \Box\big) \in V_\Box$ s.t. there exists a $\tilde{v}_1 \in V_{\mathsf{Outs}_{\Gamma(\sigma')}(v_1)}$ with $\tilde{v}_1 \succeq \hat{v}_1$, we have $\sigma(\hat{v}_1) = \sigma(\tilde{v}_1)$, when $\sigma(\tilde{v}_1) = i$ for some $i \in [n]$ and $f(\min(\mathsf{Supp}(\hat{c}_{\sigma(\tilde{v}_1)}))) = \min(\mathsf{Supp}(\hat{c}_{\sigma(\tilde{v}_1)}))$ for the injective function f defined above and $\sigma(\hat{v}_1) = \varepsilon$ otherwise. For all the remaining vertices v, we have $\sigma(v) = \sigma'(v)$. Clearly, σ is $\succeq$-compatible. From Lemma 8, since $\succeq$ is an alternating simulation relation and since an ε action can be scheduled from every $\Box$ vertex, it is always possible to construct such a strategy σ.

Now we show that σ is safe from v_2. We show by induction on the length of the play that playing σ from v_2 does not visit $\bot$. Let $\sigma(v_1) = a$, and $\sigma(v_2) = \alpha$, where $a, \alpha \in [n] \cup \{\varepsilon\}$. Let $v_1 \xrightarrow{a} v_{1,\bigcirc}$ and $v_2 \xrightarrow{\alpha} v_{2,\bigcirc}$. Recall that for all $i \in [n]$, we have $d_i^1 = d_i^2$ and $a_i^1 = a_i^2$. Further from the definition of σ, we have that $\bigcirc$ player has the same set of actions from both $v_{2,\bigcirc}$ and $v_{1,\bigcirc}$. Let $v_{2,\bigcirc} \xrightarrow{b} v_2'$ and $v_{1,\bigcirc} \xrightarrow{b} v_1'$, where b is a $\bigcirc$ action and $v_1', v_2' \in V_\Box$. Clearly, $v_1' \succeq v_2'$, and since $v_1' \neq \bot$ then $v_2' \neq \bot$ too (because $\bot$ is not comparable to any other vertex). By induction hypothesis, σ is a $\succeq$-compatible winning strategy from v_2', and we are done. ◀

A consequence of this theorem is that the set $V_{safe}^\Box = V_{safe} \cap V_\Box$ of *safe* $\Box$ *vertices* has a special structure wrt $\succeq$. Formally, $V_{safe}^\Box$ is *downward-closed*, i.e. if $v_1 \in V_{safe}^\Box$, then for all $v_2 \in V_\Box$: $v_1 \succeq v_2$ implies that $v_2 \in V_{safe}^\Box$ as well. We can thus represent $V_{safe}^\Box$ in a compact way by keeping its *maximal elements only*. Formally, for a set S of vertices, we let $\lceil S \rceil = \{v \mid \nexists v' \in S : v' \neq v \text{ and } v' \succeq v\}$ be its *maximal antichain*. When the set S is downward-closed, $\lceil S \rceil$ can be regarded as a symbolic (compact) representation of S. In the next section, we will rely on $A_{safe}^\Box = \lceil V_{safe}^\Box \rceil$.

6 Implementation and Experiments

In this section, we discuss a prototype tool implementing the techniques described so far. The tool uses the networkx library [23] to build the pruned MDP (see the steps outlined in the proof of Theorem 4 in Section 4) containing only the transitions that allow all possible schedules in which no hard task misses a deadline. This is given as an input to the Storm model-checker which analyses the MDP and finds an optimal schedule among the set of safe schedules.

We have run our prototype on a small benchmark with different numbers of hard and soft tasks. We measure the size of the system as the number of vertices in the MDP which is analysed by Storm. We ran our experiments in a MacBook Air with 1.7 GHz Intel Core i5 processor with 2 cores and having 4GB memory. We ran three different experiments:

1. We compute the time taken by our tool to remove the unsafe schedules for increasing number of hard tasks and increasing number of vertices. The results are shown in Table 1. We have only hard tasks in these experiments and the values of all the parameters is less than or equal to 12. We perform the following optimisation when constructing the set of safe vertices. We call a vertex $s = \langle s_1, \ldots, s_n, X\rangle$ with $X \in \{\Box, \bigcirc\}$ to be *immediately unsafe* if there is a task $j \in H$ s.t. $s_j = (c_j, d_j, a_j)$ with $\max(\mathsf{Supp}(c_j)) > d_j$, i.e. in

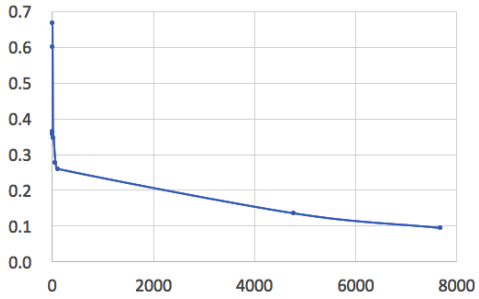

Figure 2 Fraction of vertices in the antichain (Y-axis) with increasing number of safe vertices (X-axis).

Table 1 Removing the unsafe schedules for a set of $|H|$ hard tasks. $|V|$ and $|V_{safe}|$ are the number of initial and safe vertices in the MDP respectively, T_{MDP} is the time to compute the entire MDP and T_{safe} is the time required to remove the unsafe vertices (all times in second)

	$\|H\|$	$\|V\|$	$\|V_{safe}\|$	T_{MDP}	T_{safe}
1	2	32	0	0.02	0.02
2	3	1,155	0	5.49	0.4
3	3	6,550	6,015	231.37	5.29
4	5	4,397	0	122.41	1.58
5	6	2,875	0	53.05	1.21
6	6	7,685	0	339.52	3.88

Table 2 Experiments where the number $|S|$ of soft tasks varies. $|V_{safe}|$ is the number of safe vertices in the MDP that is analysed by STORM and T is the running time of STORM to compute the mean-cost C.

	$\|S\|$	$\|V_{safe}\|$	T	C
1	1	230	0.03	0
2	2	5,369	0.39	0.07
3	3	150,895	73.09	0.28

Table 3 Experiments where the number $|H|$ of hard tasks varies. $|V_{safe}|$ is the number of safe vertices in the MDP that is analysed by STORM and T is the running time of STORM to compute the mean-cost C.

	$\|H\|$	$\|V_{safe}\|$	T	C
1	1	560	0.05	0
2	2	8,040	2.35	0
3	3	9,626	6.08	0

the rct distribution, the maximum remaining computation time exceeds the remaining time before the deadline. While computing the set of reachable vertices, once we detect that a vertex is immediately unsafe, we stop exploring from that node further. We note that both T_{MDP} and T_{safe} are proportional to the number $|V|$ of vertices rather than the number of hard tasks in the system and the number of vertices may not be directly related to the number of tasks in the system, but also depends on the parameters of the tasks. Most of these experiments were designed so that a safe schedule does not exist for the set of hard tasks considered, that is, all the vertices in V were found to be unsafe. We however have one experiment (number 3 in which most of the vertices are safe and we note that T_{safe} is not negligible in this case.

2. We run our complete algorithm on examples where all the task parameters are less than or equal to 10, we have only one hard task and the number of *soft* tasks varies, see Table 2.
3. Symmetrically, we run our complete algorithm on examples where all the task parameters are less than or equal to 9, we have only one soft task and the number of *hard* tasks varies, see Table 3.

Symbolic vertices and transitions with antichain. Finally, let us explain how we have exploited the relation defined in Section 5 in our prototype. As explained earlier, the

set $V^{\Box}_{safe}$ of *safe* $\Box$ vertices of Γ_τ is *downward-closed* for the relation $\succeq$, and can be thus represented by its maximal antichain $A^{\Box}_{safe} = \lceil V^{\Box}_{safe} \rceil$. Our experiments show that $A^{\Box}_{safe}$ is, in practice, much smaller that $V^{\Box}_{safe}$, as can be seen in Figure 2, where we plot the ratio $\frac{|A^{\Box}_{safe}|}{|V^{\Box}_{safe}|}$ against $|V^{\Box}_{safe}|$ on a set of randomly generated systems where some have only hard tasks while some are with both hard and soft tasks. The number of different tasks varies from 1 to 6. We notice that the fraction reduces with increasing number of vertices and it reduces to less than 10% when the number of safe vertices is a few thousands.

We then exploit $A^{\Box}_{safe}$ to obtain a more succinct and more structured input file to the Storm model checker. The input syntax of Storm uses a guarded command language, where the states of the system are described by means of variables that are tested and updated by a transition. So, a possible transition could be of the form:

```
rct1_1=1 & d1=3 & p1_1=4 & ... ->  (rct1_1'=1) & (d1'=2) & (p1_1'=3) & ...;
```

where `rct1_1`, `d1`, etc are the variables that encode the system state, and their primed versions characterise the successors from $V_\Box$, as the vertices from $V_\bigcirc$ are not explicitly encoded in the Storm models. In the experiments we have described above, this is how we have encoded the Storm models: all transitions from all safe states are encoded explicitly by means of conjunctions of equalities on the systems states.

Now that we have a compact representation $A^{\Box}_{safe} = \{v_1, v_2, \ldots, v_n\}$ of the safe states, we further improve the encoding of the Storm model, by testing, in the guard of the transition, whether the potential successor state v' is s.t. $v_i \succeq v'$ for some i. We then use inequalities in the guards of the transition and describe several possible transitions at once, for example:

```
rct1_1>=0 & d1=3 & ... &
((rct1_1<=1 & rct2_1-1<=2 & rct2_2-1<=4) | (rct1_1<=2 & ...) | ...)
-> (rct1_1'=rct1_1) & ...;
```

Observe that now we test that variable `rct_1` is ≥ 0 and constrain that it is unmodified (`rct1_1'=rct1_1`), so the guard is satisfied by potentially several vertices at once. The part of the guard that appears on the second line tests whether for a vertex v that is reached after the transition, there exists a $v' \in A^{\Box}_{safe}$ such that $v' \succeq v$. See Appendix C for a more detailed discussion of this new encoding. We however noted that though this approach produces a succinct input file for Storm, the latter constructs an MDP with all the vertices and transitions among them appearing explicitly in the MDP.

Another possible optimisation that our preorder $\succeq$ allows would be to compute $A^{\Box}_{safe}$ *directly* by maintaining, during the fix point computation of the attractor, sets that are antichains only, in the spirit of [20]. This has the potential to speed up the computation of the safe states. We leave this implementation for future works.

7 Discussion and Future Work

In this paper, given a set of hard and soft tasks, we described an algorithm to compute a strategy that is safe and has the minimal expected mean-cost. We formalise the construction of an MDP and show that a safe and optimal schedule can be implemented as a simple table lookup unlike many of the existing scheduling algorithm that requires more computations during scheduling the tasks. We also implemented a prototype of a tool and use Storm model-checker to show that our approach can indeed be used in practice.

Using reinforcement learning. Our algorithm relies on a stochastic model for the arrival of the tasks and the duration of the tasks. If this stochastic model is not available, techniques like reinforcement learning can be used for the online construction of a stochastic model during interaction with the tasks. The challenge here is to combine reinforcement learning with techniques to maintain the safety for the deadline of the hard tasks. Algorithms to combine learning with hard guarantees are currently explored [44].

References

1 L. Abeni and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–13, Washington, DC, USA, 1998. IEEE Computer Society.

2 L. Abeni and G. C. Buttazzo. Stochastic Analysis of a Reseveration Based System. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, April 23-27, 2001*, pages 92–98, 2001.

3 S. Almagor, O. Kupferman, and Y. Velner. Minimizing Expected Cost Under Hard Boolean Constraints, with Applications to Quantitative Synthesis. In *Proc. 27th Int. Conf. on Concurrency Theory*, volume 59 of *LIPIcs*, pages 9:1–9:15, 2016.

4 R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *Proc. 9th Int. Conf. on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998.

5 A. Atlas and A. Bestavros. Statistical Rate Monotonic Scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 123–132, 1998.

6 H. Aydin, R. G. Melhem, D. Mossé, and P. Mejía-Alvarez. Optimal Reward-Based Scheduling of Periodic Real-Time Tasks. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*, pages 79–89, 1999.

7 S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990*, pages 182–190, 1990.

8 S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. *Real-Time Systems*, 2(4):301–324, 1990.

9 V. Bruyère, E. Filiot, M. Randour, and J-F. Raskin. Meet Your Expectations With Guarantees: Beyond Worst-Case Synthesis in Quantitative Games. In *Proc. 31th Symp. on Theoretical Aspects of Computer Science*, volume 25 of *LIPIcs*, pages 199–213, 2014.

10 G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

11 H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Trans. Software Eng.*, 15(10):1261–1269, 1989.

12 L. Clemente and J-F. Raskin. Multidimensional beyond Worst-Case and Almost-Sure Problems for Mean-Payoff Objectives. In *Proc. 30th IEEE Symp. on Logic in Computer Science*, pages 257–268, 2015.

13 C. Dehnert, S. Junges, J-P. Katoen, and M. Volk. A Storm is Coming: A Modern Probabilistic Model Checker. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, Proceedings, Part II*, pages 592–600, 2017.

14 S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Oper. Res.*, 26(1):127–140, February 1978.

15 J. L. Díaz, D. F. García, K. Kim, C-G. Lee, L. L. Bello, J. M. López, S. L. Min, and O. Mirabella. Stochastic Analysis of Periodic Real-Time Systems. In *Proceedings of the*

23rd IEEE Real-Time Systems Symposium, RTSS '02, pages 289–300, Washington, DC, USA, 2002. IEEE Computer Society.

16 L. Doyen and J.-F. Raskin. Antichains for the Automata-Based Approach to Model-Checking. *Logical Methods in Computer Science*, 5(1), 2009.

17 J. Filar and K. Vrieze. *Competitive Markov decision processes*. Springer, 1997.

18 M. K. Gardner. *Probabilistic Analysis and Scheduling of Critical Soft Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999.

19 M. K. Gardner and J. W.-S. Liu. Analyzing Stochastic Fixed-Priority Real-Time Systems. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 44–58, London, UK, UK, 1999. Springer-Verlag.

20 G. Geeraerts, J. Goossens, T-V-A Nguyen, and A. Stainer. Synthesising succinct strategies in safety games with an application to real-time scheduling. *Theoretical Computer Science*, 735:24–49, 2018.

21 John Gill. Computational Complexity of Probabilistic Turing Machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

22 C. Haase and S. Kiefer. The complexity of the Kth largest subset problem and related problems. *Information Processing Letters*, 116(2):111–115, 2016.

23 A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring Network Structure, Dynamics, and Function Using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, 2008.

24 K. S. Hong and J. Y.-T. Leung. On-Line Scheduling of Real-Time Tasks. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88), December 6-8, 1988, Huntsville, Alabama, USA*, pages 244–250, 1988.

25 J. P. Lehoczky, L. Sha, and I. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Real-Time Systems Symposium*, pages 261–270, 1987.

26 J. Y.-T. Leung and M. L. Merrill. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118, 1980.

27 J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Eval.*, 2(4):237–250, 1982.

28 K-J Lin, S. Natarajan, and J. W.-S. Liu. Imprecise Results: Utilizing Partial Comptuations in Real-Time Systems. In *Proceedings of the 8th IEEE Real-Time Systems Symposium (RTSS '87), December 1-3, 1987, San Jose, Califorinia, USA*, pages 210–217, 1987.

29 C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

30 J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao. *Algorithms for Scheduling Imprecise Computations*, pages 203–249. Springer US, Boston, MA, 1991.

31 Y. Lu, T. Nolte, I. Bate, and L. Cucu-Grosjean. A Statistical Response-Time Analysis of Real-Time Embedded Systems. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS 2012, San Juan, PR, USA, December 4-7, 2012*, pages 351–362, 2012.

32 Y. Lu, T. Nolte, J. Kraft, and C. Norström. A Statistical Approach to Response-Time Analysis of Complex Embedded Real-Time Systems. In *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2010, Macau, SAR, China, 23-25 August 2010*, pages 153–160, 2010.

33 D. Maxim, R. I. Davis, L. Cucu-Grosjean, and A. Easwaran. Probabilistic analysis for mixed criticality systems using fixed priority preemptive scheduling. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS 2017, Grenoble, France, October 04 - 06, 2017*, pages 237–246, 2017.

34 A. K. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

35 C. H. Papadimitriou and J. N. Tsitsiklis. The Complexity of Markov Decision Processes. *Math. Oper. Res.*, 12(3):441–450, 1987.

36 M.L. Puterman. *Markov Decision Processes*. Wiley, 1994.

37 J. Simon. *On Some Central Problems in Computational Complexity*. PhD thesis, Cornell University, Ithaca, NY, USA, 1975.

38 B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for Hard-Real-Time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

39 M. Spuri and G. Buttazzo. Efficient Aperiodic Service Under Earliest Deadline Scheduling. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS '94), San Juan, Puerto Rico, December 7-9, 1994*, pages 2–11, 1994.

40 M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

41 A. Srinivasan and J. H. Anderson. Efficient Scheduling of Soft Real-time Applications on Multiprocessors. *J. Embedded Comput.*, 1(2):285–302, April 2005.

42 Wolfgang Thomas. On the Synthesis of Strategies in Infinite Games. In *STACS*, pages 1–13, 1995. doi:10.1007/3-540-59042-0_57.

43 T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu. Probabilistic Performance Guarantee for Real-time Tasks with Varying Computation Times. In *Proceedings of the Real-Time Technology and Applications Symposium*, RTAS '95, pages 164–173, Washington, DC, USA, 1995. IEEE Computer Society.

44 M. Wen and U. Topcu. Probably Approximately Correct Learning in Stochastic Games with Temporal Logic Specifications. In *Proc. 25th Int. Joint Conf. on Artificial Intelligence*, pages 3630–3636. IJCAI/AAAI Press, 2016.

A Attractor algorithm for Safety Synthesis

The algorithm consists of computing all the vertices from which $\Box$ *cannot avoid* reaching the *unsafe vertices*. To this end, the algorithm computes a sequence of sets of vertices $(A_i)_{i\geq 0}$ defined as follows:

(i) $A_0 = V \setminus V_{\mathsf{safe}}$; and

(ii) for all $i \geq 0$: $A_{i+1} = A_i \cup \{v \in V_\Box \mid \mathsf{Succ}(v) \subseteq A_i\} \cup \{v \in V_\bigcirc \mid \mathsf{Succ}(v) \cap A_i \neq \emptyset\}$.

That is, the sequence $(A_i)_{i\geq 0}$ is initialised to the set of *unsafe* vertices. Then, the algorithm grows this set of vertices by adding:

(i) vertices belonging to $\Box$ whose set of successors has been entirely identified as unsafe in a previous step; and

(ii) vertices belonging to $\bigcirc$ having at least one unsafe successor.

It is easy to check that this sequence converges after at most $|V|$ steps (the graph of the MDP being finite) and returns the set of vertices $\mathsf{Attr}(V \setminus V_{\mathsf{safe}})$ from which $\Box$ has no strategy to stay within V_{safe}. Hence, $\Box$ has a strategy $\sigma_\Box$ to stay within V_{safe} from all vertices in which is s.t. $\sigma_\Box(v) \notin \mathsf{Attr}(V \setminus V_{\mathsf{safe}})$ (any successor of v satisfying this criterion yields a safe strategy).

B Comparison with Relevant Scheduling Algorithms

Several works in the literature consider real-time scheduling of systems with both soft and hard tasks. A prominent line of works among them is based on the notion of *servers* [10, 40] to handle soft tasks. Algorithms for preemptive uniprocessor scheduling following this approach

include Priority Exchange [40, 25], Sporadic Server [40, 38], Total Bandwidth Server [40], Earliest Deadline Late (EDL) Server [11, 39, 40], Constant Bandwidth Server [1], etc. Their performance are measured according to the *responsiveness* to each of the soft task requests without compromising the schedulability of the hard tasks. The responsiveness is defined as the difference between the time of completion of a request and the time of its arrival in the system. However, those algorithms do not take into account a stochastic model of the tasks as in our problem nor a notion of deadline and cost for the soft tasks. Hence they do not provide solutions to the problem that we consider in this paper.

The algorithm EDL is known to be optimal for dynamic priority assignment [11]. An EDL server algorithm is a dynamic slack-stealing algorithm in which the active periodic tasks are processed as late as possible. The basic idea behind the EDL server is to use the idle times of an EDL schedule to execute aperiodic requests (soft tasks) as soon as possible. When there are no aperiodic activities in the system, periodic tasks are scheduled according to the earliest deadline first (EDF) algorithm. An important property of EDL is that it guarantees the maximum available idle time that is used for an optimal server mechanism for soft aperiodic activities.

In order to evaluate the potential of those solutions for our problem, we consider the following modified version of EDL:

- The hard tasks are scheduled as late as possible following an EDF among the hard tasks, without compromising their schedulability.
- At every time, when a hard task is not scheduled, the active soft tasks are also scheduled according to EDF.

The algorithm is preemptive as in the original setting.

The example below shows that in our setting, the ratio of the expected costs obtained with the modified EDL and the the expected cost of the optimal strategy can be arbitrarily large.

▶ **Example 10.** Consider a system with three tasks: one hard task h and two soft tasks s_1 and s_2. The hard task h has execution time, deadline and a period of 2, 3 and 3 respectively. The execution time, deadline and period of s_1 are 1, 3 and 3 respectively. For s_2, the execution time and the deadline are 1 and 2 respectively while the inter-arrival time is 3 with probability 0.1 and 6 with probability 0.9. Let the cost of missing the deadline for an instance of s_1 and s_2 be c_1 and c_2 respectively and let $c_2 > c_1$. The first instance of each of h and s_1 arrives at time 0 while the first instance of s_2 arrives at time 1. We divide the entire timeline into *blocks* of 3 time units with consecutive odd and even blocks and the first block is an odd block. In every even block, an instance of task s_2 appears with probability 0.1 while in every odd block excluding the first block, it appears with probability 0.9. Thus in the optimal strategy, in every even block, s_1 misses its deadline with probability 0.1 while in every odd block it misses its deadline with probability 0.9. The optimal schedule prioritises scheduling s_2 over scheduling s_1 since $c_2 > c_1$. A strategy that produces the minimum expected mean cost thus has an associated cost proportional to $0.1 \cdot c_1 + 0.9 \cdot c_1 = c_1$.

The modified version of the EDL, on the other hand, in both odd and even blocks, schedules the soft task s_1. Thus with probability 0.1, an instance of the soft task s_2 misses its deadline in every even interval and with probability 0.9, it misses its deadline in every odd interval and hence the minimum expected mean cost is proportional to c_2.

As we increase c_2, we see that the ratio of the costs obtained from using the modified EDL and the optimal strategy cannot be bounded above.

Note that the above example could be simplified by considering a Dirac distribution on the inter-arrival time of task s_2 that can be set to 3. However, the example above illustrates the robustness of our approach in the sense that it can find an optimal schedule even when we consider arbitrary probability distributions. The optimal schedule generated by our approach indeed changes with the change in the distribution. We thus have the following proposition.

▶ **Proposition 11.** *There exists a family of systems S in which the ratio of the expected mean costs of missing the deadlines of the soft tasks by using the modified EDL and the optimal strategy (as obtained by our algorithm) can be arbitrarily large.*

Further, EDL suffers from the following problems. Although optimal with respect to the finishing time of the soft tasks, it involves a heavy overload for the computation of the idle times (slacks) that makes it less practical [10]. As seen in Proposition 11, since EDL does not consider any cost associated to missing deadlines of the soft tasks, it is not optimal for our setting. Furthermore, it does not take advantage of the information that is given by the stochastic model of the tasks.

We also consider a simple adaptation of the EDF algorithm to schedule hard and soft tasks to our setting that we call the two- stage EDF.

Two-stage EDF. The two-stage EDF is described as follows. Among the set of safe strategies, first the hard tasks are scheduled by EDF strategy. Next the soft tasks are scheduled by EDF strategy only when there does not exist an active hard task in the system. We show that in the worst case, the two-stage EDF algorithm can be arbitrarily bad in terms of the mean cost.

▶ **Proposition 12.** *There exists a family of systems S in which the ratio of the expected mean costs of missing the deadlines of the soft tasks by using the two-stage EDF algorithm and the optimal strategy (as obtained by our algorithm) can be arbitrarily large.*

Proof. Consider a deterministic system with one hard and one soft task. Let the execution time, deadline and period of the hard task be $1, 2$ and 2 respectively. The parameters for the soft task are $1, 1$ and 2 respectively. The first instance of both the hard and the soft task arrives the system at time 0 and let the cost of missing the deadline for each job of the soft task be 20. The two-phase EDF first schedules the hard task and thus for every instance of the soft task, it misses the deadline. Since the period is 2, the expected mean cost for missing the deadline for the soft task instances is 10.

On the other hand, the optimal schedule generated by our approach always schedules the soft task before the hard task and hence the soft task always finishes execution before the deadline and hence the mean cost is 0 and we are done. ◀

We also have an implementation of the two-stage-EDF and compare the time taken to compute the mean-cost by our optimal algorithm and the two-stage EDF. Given a specific schedule (two-stage EDF in our experiments) it corresponds to a fixed strategy of Scheduler, and hence the state space of the MDP that is analysed by Storm to compute the expected mean-cost for this particular strategy is usually only a small part entire state space of the MDP that is given as an input to Storm. Computing the cost corresponding to an optimal schedule giving the minimum expected mean-cost, on the other hand, requires Storm to find the corresponding optimal strategy of Scheduler which involves exploring the entire state space of this MDP.

In Table 4, $|Soft|$ denotes the number of soft tasks, S_{EDF} denotes the number of vertices in the system that is analysed by Storm when we consider the two-stage EDF schedule and T_{EDF} is the time required to analyse the system for two-stage EDF and C_{EDF} denotes the

Table 4 Comparison between two-stage EDF and our optimal algorithm with different number of soft tasks.

	$\|Soft\|$	S_{EDF}	S_{OPT}	T_{EDF}	T_{OPT}	C_{EDF}	C_{OPT}
1	1	71	230	0.03	0.03	0.18	0
2	2	876	5369	0.08	0.39	0.22	0.07
3	3	18273	150895	4.93	73.09	0.68	0.28

Table 5 Comparison between two-stage EDF and our optimal algorithm with different number of hard tasks.

	$\|Hard\|$	S_{EDF}	S_{OPT}	T_{EDF}	T_{OPT}	C_{EDF}	C_{OPT}
1	1	109	560	0.04	0.05	0.07	0
2	2	810	8040	0.38	2.35	0.26	0
3	3	808	9626	0.93	6.08	0.24	0

mean-cost that we obtain for two-stage EDF. Similarly, we have S_{OPT}, T_{OPT} and C_{OPT} for our optimal algorithm. The columns S_{OPT}, T_{OPT} and C_{OPT} are the same as S_{safe}, T and C in Table 2 and Table 3 respectively. In these experiments, all the parameters, that is, the supports in the execution time distribution, the deadline and the supports in the inter-arrival time distribution for every task have values less than or equal to 10.

In Table 5, the first column $|Hard|$ denotes the number of hard tasks in the system. We use one soft task. The trends of the results are similar to those in Table 4. The values of all the parameters used in these experiments is less than or equal to 9.

Further, there have been several studies to analyse quality of service driven applications. These studies use stochastic tools to analyse the execution times of different tasks and their effects on the quality of service [15, 43, 18, 19, 5, 2, 32, 31, 33]. Those methods do not propose synthesis techniques and consider that the scheduler preexists.

C Optimising the Storm input file with antichains

An example of a transition using concrete vertices is like the following. Here every state is represented concretely. `[hard2]` is the task that is executed, that is, it corresponds to the second hard task as specified in the input file.

```
[hard2] rct1_1=1 & d1=3 & p1_1=4 & p1_2=5 &
rct2_1=3 & rct2_2=5 & d2=5 & p2_1=6 & p2_2=7 ->
(rct1_1'=1) & (d1'=2) & (p1_1'=3) & (p1_2'=4) &
(rct2_1'=2) & (rct2_2'=4) & (d2'=4) & (p2_1'=5) & (p2_2'=6);
```

The variable `[rct1_1]` denotes the first element in the support of the rct distribution of task τ_1 when the elements of the support are arranged in an increasing order, while `[rct2_1]` denotes the first element of the support in the rct distribution of task τ_2 and so on. `d1` is the remaining deadline of task τ_1, and `d2` denotes the remaining deadline for task τ_2. `p1_1` and `p1_2` respectively denote the first and the second elements in an increasing order in the support of the remaining time before the arrival of the next job for task τ_1. The part of the transition on the left side of `->` is the guard while the part on its right is the new state reached. Note that in this particular example, we have only state that is reached following the transition.

A transition using symbolic states is the following.

```
[hard2] rct1_1>=0 & d1=3 & p1_1=4 & p1_2=5 & rct2_1>=2 &
rct2_2>=4 & d2=5 & p2_1=6 & p2_2=7 &
((rct1_1<=1 & rct2_1-1<=2 & rct2_2-1<=4) |
(rct1_1<=2 & rct2_1-1<=1 & rct2_2-1<=3)) ->
(rct1_1'=rct1_1) & (d1'=2) & (p1_1'=3) & (p1_2'=4) &
(rct2_1'=rct2_1-1) & (rct2_2'=rct2_2-1) & (d2'=4) & (p2_1'=5) & (p2_2'=6);
```

Note that in the rct distributions, we have inequalities instead of equalities and hence transitions corresponding to several concrete states are represented by this. The part `((rct1_1<=1 & rct2_1-1<=2 & rct2_2-1<=4) | (rct1_1<=2 & rct2_1-1<=1 & rct2_2-1<=3))` denotes that the state reachable following the transition should be less difficult than at least one of the two elements of the antichain. Each disjunct corresponds to a comparison with an element of the antichain.