Universiteit
Antwerpen

This item is the archived peer-reviewed author-version of:

Safe learning for near-optimal scheduling

**Reference:**
Busatto-Gaston Damien, Chakraborty Debraj, Guha Shibashis, Pérez Guillermo Alberto, Raskin Jean-François.- Safe learning for near-optimal scheduling
Lecture notes in computer science - ISSN 1611-3349 - Springer, 12846(2021), p. 235-254
Full text (Publisher's DOI): https://doi.org/10.1007/978-3-030-85172-9_13
To cite this reference: https://hdl.handle.net/10067/1801190151162165141

uantwerpen.be
Institutional repository IRUA

# Safe Learning for Near-Optimal Scheduling*

Damien Busatto-Gaston[1 [0000-0002-7266-0927]], Debraj Chakraborty[1 [0000-0003-0978-4457]], Shibashis Guha[2 [0000-0002-9814-6651]], Guillermo A. Pérez[3 [0000-0002-1200-4952]], and Jean-François Raskin[1 [0000-0002-3673-1097]]

[1] Université libre de Bruxelles, Belgium
[2] Tata Institute of Fundamental Research, India
[3] University of Antwerp – Flanders Make, Belgium

**Abstract.** In this paper, we investigate the combination of synthesis, model-based learning, and online sampling techniques to obtain safe and near-optimal schedulers for a preemptible task scheduling problem. Our algorithms can handle Markov decision processes (MDPs) that have $10^{20}$ states and beyond which cannot be handled with state-of-the art probabilistic model-checkers. We provide probably approximately correct (PAC) guarantees for learning the model. Additionally, we extend Monte-Carlo tree search with advice, computed using safety games or obtained using the earliest-deadline-first scheduler, to safely explore the learned model online. Finally, we implemented and compared our algorithms empirically against shielded deep $Q$-learning on large task systems.

**Keywords:** Model-based learning · Monte-Carlo tree search · Task scheduling

## 1 Introduction

In this paper, we show how to combine synthesis, model-based learning, and online sampling techniques to solve a scheduling problem featuring both hard and soft constraints. We investigate solutions to this problem both from a theoretical and from a more pragmatic point of view. On the theoretical side, we show how safety guarantees (as understood in formal verification) can be combined with guarantees offered by the probably approximately correct (PAC) learning framework [23]. On the pragmatic side, we show how safety guarantees obtained from automatic synthesis can be combined with Monte-Carlo tree search (MCTS) [20] to offer a scalable and practical solution to solve the scheduling problem at hand.

The scheduling problem that we consider is defined as follows. A task system is composed of a set of $n$ preemptible tasks $(\tau_i)_{i\in[n]}$ partitioned into a set $F$ of soft tasks and a set $H$ of hard tasks. Time is assumed to be discrete and measured e.g. in CPU ticks. Each task $\tau_i$ generates an infinite number of instances $\tau_{i,j}$,

* This work was supported by the ARC "Non-Zero Sum Game Graphs" project (Fédération Wallonie-Bruxelles), the EOS "Verilearn" project (F.R.S.-FNRS & FWO), and the FWO "SAILor" project (G030020N).

called *jobs*, with $j = 1, 2, \ldots$ 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. The computation time requirements of the jobs follow a discrete probability distribution, and are unknown to the scheduler but upper bounded by their relative deadline. Jobs generated by hard tasks must complete before their respective deadlines. For jobs generated by soft tasks, deadline misses result in a penalty/cost. The tasks are assumed to be independent and generated stochastically: the occurrence of a new job of one task does not depend on the occurrences of jobs of other tasks, and both the inter-arrival and computation times of jobs are independent random variables. The scheduling problem consists in finding a *scheduler*, i.e. a function that associates, to all CPU ticks, a task that must run at that moment; in order to: (i) avoid deadline misses by hard tasks; and (ii) minimise the mean cost of deadline misses by soft tasks.

In [13], we modelled the semantics of the task system using a Markov decision process (MDP) and posed the problem of computing an optimal and safe scheduler. However, that work assumes that the distribution of all tasks is known a priori which may be unrealistic. Here, we investigate learning techniques to build algorithms that can schedule safely and optimally a set of hard and soft tasks if only the deadlines and the domains of the distributions describing the tasks of the system are known a priori and not the exact distributions. This is a more realistic assumption. Our motivation was also to investigate the joint application of both synthesis techniques coming from the field of formal verification and learning techniques on an understandable yet challenging setting.

**Contributions.** First, we show the distributions underlying a task system with only soft tasks are *efficiently* PAC learnable: by executing the task system for a polynomial number of steps, enough samples can be collected to infer $\varepsilon$-accurate approximations of the distributions with high probability (Thm. 1).

Then, we consider the general case of systems with both hard and soft tasks. Here, safe PAC learning is *not* always possible, and we identify two algorithmically-checkable sufficient conditions for task systems to be safely learnable (Thms. 2 and 3). These crucially depend on the underlying MDP being a single maximal end-component, as is the case in our setting (Lem. 2). Subsequently, we can use *robustness* results on MDPs to compute or learn near-optimal safe strategies from the learnt models (Thm. 4).

Third, in order to evaluate the relevance of our algorithms, we present experiments of a prototype implementation. These empirically validate the efficient PAC guarantees. Unfortunately, the learnt models are often too large for the probabilistic model-checking tools. In contrast, the MCTS-based algorithm scales to larger examples: e.g. we learn safe scheduling strategies for systems with more than $10^{20}$ states. Our experiments also show that a strategy obtained using deep $Q$-learning [2,18] by assigning high costs to missing deadlines of hard tasks does not respect safety, even if one learns for a long period of time and the deadline-miss costs of hard tasks are very high (cf. [1]).

**Related works** In [13], we introduced the scheduling problem considered here but made the assumption that the underlying distributions of the tasks are

known. We drop this assumption here and provide learning algorithms. In [1], the framework to combine safety via shielding and model-free reinforcement learning is introduced and applied to several examples using table-based Q-learning as well as deep RL. In [3], shield synthesis is studied for long-run objective guarantees instead of safety requirements. Unlike our work, the transition probabilities on MDPs in both [1] and [3] are assumed to be known. We observe that [1] and [3] do not provide model-based learning and PAC guarantees. While some pre-shielding literature does consider unknown MDPs (see, e.g.[12]), we are not aware of PAC-learning works that focus on scheduling problems.

In [16], we studied a framework to mix reactive synthesis and model-based reinforcement learning for mean-payoff with PAC guarantees. There, the learning algorithm estimates the probabilities on the transitions of the MDP. In our approach, we do not estimate these probabilities directly from the MDP, but learn probabilities for the individual tasks in the task system. The efficient PAC guarantees that we have obtained for the model-based part cannot be obtained from that framework. Finally, in [8] we introduced a first combination of shielding with model-predictive control using MCTS, but did not consider learning.

## 2 Preliminaries

We denote by $\mathbb{N}$ the set of natural numbers; by $\mathbb{Q}$, the set of rational numbers; and by $\mathbb{Q}_{\geq 0}$ the set $\{q \in \mathbb{Q} \mid q \geq 0\}$ of all non-negative rational numbers. Given $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, \ldots, n\}$. 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 call $A$ the *domain of* $p$, and denote it by $\mathsf{Dom}(p)$. 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 $|\mathsf{Supp}(p)| = 1$. For a probability distribution $p$, the minimum probability assigned by $p$ to the elements in $\mathsf{Supp}(p)$ is $\pi^p_{\min} = \min\limits_{a \in \mathsf{Supp}(p)} (p(a))$. We say two distributions $p$ and $p'$ are *structurally identical* if $\mathsf{Supp}(p) = \mathsf{Supp}(p')$. Given two structurally identical distributions $p$ and $p'$, for $0 < \varepsilon < 1$, we say that $p$ is $\varepsilon$-close to $p'$, denoted $p \sim^{\varepsilon} p'$, if $\mathsf{Supp}(p) = \mathsf{Supp}(p')$, and for all $a \in \mathsf{Supp}(p)$, we have that $|p(a) - p'(a)| \leq \varepsilon$.

**Scheduling problem** An instance of the scheduling problem studied in [13] consists of a task system $\Upsilon = ((\tau_i)_{i \in [n]}, F, H)$, where $(\tau_i)_{i \in [n]}$ are $n$ preemptible tasks partitioned into hard and soft tasks $H$ and $F$ respectively. The latter need to be scheduled on a *single processor*. Formally, the work of [13] relies on a probabilistic model for the computation times of the jobs and for the delay between the arrival of two successive jobs of the same task. For all $i \in [n]$, task $\tau_i$ is defined as a tuple $\langle \mathcal{C}_i, D_i, \mathcal{A}_i \rangle$, where: (i) $\mathcal{C}_i$ is a discrete probability distribution on the (finitely many) possible computation times of the jobs generated by $\tau_i$; (ii) $D_i \in \mathbb{N}$ is the deadline of all jobs generated by $\tau_i$ which is relative to their arrival time; and (iii) $\mathcal{A}_i$ is a discrete probability distribution on the (finitely many) possible inter-arrival times of the jobs generated by $\tau_i$. We denote by $\pi^{\Upsilon}_{\max}$ the maximum probability appearing in the definition of $\Upsilon$, that is, across all the distributions $\mathcal{C}_i$ and $\mathcal{A}_i$, for all $i \in [n]$. It is assumed that

$\max(\mathsf{Dom}(\mathcal{C}_i)) \leq D_i \leq \min(\mathsf{Dom}(\mathcal{A}_i))$ for all $i \in [n]$; hence, at any point in time, there is at most one job per task in the system. Also note that when a new job of some task arrives at the system, the deadline for the previous job of this task is already over. Finally, we assume that the task system is *schedulable for the hard tasks*, meaning that it is possible to guarantee that jobs associated to hard tasks never miss their deadlines. On the other hand, the full set of tasks may not be schedulable, so that jobs associated with soft tasks may be allowed to miss their deadlines. The potential degradation in the quality when a soft task misses its deadline is modelled by a cost function $cost : F \to \mathbb{Q}_{\geq 0}$ that associates to each soft task $\tau_j$ a cost $c(j)$ that is incurred every time a job of $\tau_j$ misses its deadline. As a final observation, we recall the *earliest deadline first* (EDF) algorithm that always gives execution time to the job closest to its deadline. EDF is an optimal scheduling algorithm in the following sense: if a task system is schedulable (without any misses at all) then EDF will yield such a feasible schedule [6]. In general, applying EDF on both the hard and soft tasks may cause hard tasks to miss deadlines, as the entire task system may not be schedulable. However, one may apply EDF on hard tasks only, and allow for soft tasks whenever no hard task is available. This version of EDF ensures that all jobs of hard tasks are scheduled in time, but does not guarantee optimality with respect to cost.

Given a task system $\Upsilon = ((\tau_i)_{i\in[n]}, F, H)$ with $n$ tasks, the structure of $\Upsilon$ is $((\mathsf{struct}(\tau_i))_{i\in[n]}, F, H)$ where $\mathsf{struct}(\langle \mathcal{C}, D, \mathcal{A}\rangle) = (\langle \mathsf{Dom}(\mathcal{C}), D, \mathsf{Dom}(\mathcal{A})\rangle)$. We denote by $\mathcal{C}_{\max}$ and $\mathcal{A}_{\max}$ resp. the maximum computation time, and the maximum inter-arrival time of a task in $\Upsilon$. Formally, $\mathcal{C}_{\max} = \max(\bigcup_{i\in[n]} \mathsf{Dom}(\mathcal{C}_i))$, and $\mathcal{A}_{\max} = \max(\bigcup_{i\in[n]} \mathsf{Dom}(\mathcal{A}_i))$. Note that $\mathcal{A}_{\max} \geq \mathcal{C}_{\max}$. We also let $\mathbb{D} = \max_{i\in[n]}(|\mathsf{Dom}(\mathcal{A}_i)|)$. We denote by $|\Upsilon|$ the number of tasks in the task system $\Upsilon$. Consider two task systems $\Upsilon_1 = ((\tau_i^1)_{i\in[n]}, F, H)$, and $\Upsilon_2 = ((\tau_i^2)_{i\in[n]}, F, H)$, with $|\Upsilon_1| = |\Upsilon_2|$, $\tau_i^j = \langle \mathcal{C}_i^j, D_i^j, \mathcal{A}_i^j\rangle$ for all $i \in [n]$ and $j \in [2]$. The two task systems $\Upsilon_1$ and $\Upsilon_2$ are said to be *$\varepsilon$-close*, denoted $\Upsilon_1 \approx^\varepsilon \Upsilon_2$, if (i) $\mathsf{struct}(\Upsilon^1) = \mathsf{struct}(\Upsilon^2)$, (ii) for all $i \in [n]$, we have $\mathcal{A}_i^1 \sim^\varepsilon \mathcal{A}_i^2$, and (iii) for all $i \in [n]$, we have $\mathcal{C}_i^1 \sim^\varepsilon \mathcal{C}_i^2$.

**Markov decision processes** Let us now introduce *Markov Decision Process* (MDP) as they form the basis of the formal model of [13], which we recall later. A finite *Markov decision process* is a tuple $\Gamma = \langle V, E, L, (V_\square, V_\bigcirc), A, \delta, \mathsf{cost}\rangle$, where: (i) $A$ is a finite set of actions; (ii) $\langle V, E\rangle$ is a finite directed graph and $L$ is an edge-labelling function (we denote by $E(v)$ the set of outgoing edges from vertex $v$); (iii) the set of vertices $V$ is partitioned into $V_\square$ and $V_\bigcirc$; (iv) the graph is bipartite i.e. $E \subseteq (V_\square \times V_\bigcirc) \cup (V_\bigcirc \times V_\square)$, and the labelling function is s.t. $L(v, v') \in A$ if $v \in V_\square$, and $L(v, v') \in \mathbb{Q}$ if $v \in V_\bigcirc$; and (v) $\delta$ 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 $v \in V_\square$, for all $e$, $e'$ in $E(v)$: $L(e) = L(e')$ implies $e = e'$, i.e. an action identifies uniquely an outgoing edge. Given $v \in V_\square$, and $a \in A$, we define $\mathsf{Post}(v, a) = \{v' \in V_\bigcirc \mid (v, v') \in E \text{ and } L(v, v') = a\} \cup \{v'' \in V_\square \mid \exists v' : (v, v') \in E, L(v, v') = a \text{ and } \delta(v', v'') > 0\}$. For all vertices $v \in V_\square$, we denote by $A(v)$, the set of actions $\{a \in A \mid \mathsf{Post}(v, a) \cap V_\square \neq \emptyset\}$. The size of an MDP $\Gamma$, denoted $|\Gamma|$, is the sum of the number of vertices and the number of edges, that

is, $|V| + |E|$. An MDP $\Gamma = \langle V, E, L, (V_\square, V_\bigcirc), A, \delta, \mathsf{cost}\rangle$ is said to *structurally identical* to another MDP $\Gamma' = \langle V, E, L', (V_\square, V_\bigcirc), A, \delta', \mathsf{cost}\rangle$ if for all $v \in V_\bigcirc$, we have that $\mathsf{Supp}(\delta(v)) = \mathsf{Supp}(\delta'(v))$. For two structurally identical MDPs $\Gamma$ and $\Gamma'$ with distribution assignment functions $\delta$ and $\delta'$ respectively, we say that $\Gamma$ is $\varepsilon$-approximate to $\Gamma'$, denoted $\Gamma \approx^\varepsilon \Gamma'$, if for all $v \in V_\bigcirc$: $\delta(v) \sim^\varepsilon \delta'(v)$.

An MDP $\Gamma$ can be interpreted as a game $\mathcal{G}_\Gamma$ between two players: $\square$ and $\bigcirc$, who own the vertices in $V_\square$ and $V_\bigcirc$ respectively. A play in an MDP is a path in its underlying graph $\langle V, E, A \cup \mathbb{Q}\rangle$. We say that a prefix $\pi(n)$ of a play $\pi$ belongs to player $i \in \{\square, \bigcirc\}$, iff its last vertex $\mathsf{Last}(\pi(n))$ is in $V_i$. The set of prefixes that belong to player $i$ is denoted by $\mathsf{Prefs}_i(\mathcal{G}_\Gamma)$. A play is obtained by the interaction of the players: if the current play prefix $\pi(n)$ belongs to $\square$, 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 $\delta(\mathsf{Last}(\pi(n)))$. In both cases, the plays prefix is extended by $e$ and the game goes *ad infinitum*.

A (deterministic) *strategy* of $\square$ is a function $\sigma_\square : \mathsf{Prefs}_\square(\mathcal{G}) \to E$, such that $\sigma_\square(\rho) \in E(\mathsf{Last}(\rho))$ for all prefixes. A strategy $\sigma_\square$ is *memoryless* if for all finite prefixes $\rho_1$ and $\rho_2 \in \mathsf{Prefs}(\mathcal{G})$: $\mathsf{Last}(\rho_1) = \mathsf{Last}(\rho_2)$ implies $\sigma_\square(\rho_1) = \sigma_\square(\rho_2)$. For memoryless strategies, we will abuse notations and assume that such strategies $\sigma$ are of the form $\sigma : V_\square \to E$ (i.e., the strategy associates the edge to play to the current vertex and not to the full prefix played so far). From now on, we will consider memoryless deterministic strategies unless otherwise stated. Let $\Gamma = \langle V, E, L, (V_\square, V_\bigcirc), A, \delta, \mathsf{cost}\rangle$ be an MDP, and let $\sigma_\square$ be a *memoryless* strategy. Then, assuming that $\square$ plays according to $\sigma_\square$, we can express the behaviour of $\Gamma$ as a Markov chain $\Gamma[\sigma_\square]$, where the probability distributions reflect the stochastic choices of $\bigcirc$ (see [13] for the details).

**End components** An *end-component* (EC) $M = (T, A')$, with $T \subseteq V$ and $A' : T \cap V_\square \to 2^A$, is a *sub-MDP* of $\Gamma$ such that: for all $v \in T \cap V_\square$, $A'(v)$ is a subset of the actions available to $\square$ from $v$; for all $a \in A'(v)$, $\mathsf{Post}(v, a) \subseteq T$; and, it's underlying graph is strongly connected. A *maximal end-component* (MEC) is an EC that is not included in any other EC.

**MDP for the scheduling problem** Given a system $\Upsilon = \{\tau_1, \tau_2, \dots, \tau_n\}$ of tasks, we describe below the modelling of the scheduling problem by an MDP $\Gamma_\Upsilon = \langle V, E, L, (V_\square, V_\bigcirc), A, \delta, \mathsf{cost}\rangle$ as it appears in [13]. The two players $\square$ and $\bigcirc$ correspond respectively to the *Scheduler* and the task generator (*TaskGen*) respectively. Since there is at most one job per task that is active at all times, vertices encode the following information about each task $\tau_i$: (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. We also tag vertices with either $\square$ or $\bigcirc$ to remember their respective owners and we have a vertex $\bot$ that is reached when a hard task misses a deadline. For a vertex $v = \big((c_1, d_1, a_1) \dots (c_n, d_n, a_n), \Delta\big)$, for $\Delta \in \{\square, \bigcirc\}$, let $\mathsf{active}(v) = \{i \mid c_i(0) \neq 1 \text{ and } d_i > 0\}$ be the tasks that have an active job in $v$; $\mathsf{dlmiss}(v) = \{i \mid c_i(0) = 0 \text{ and } d_i = 0\}$, those that have missed a deadline in $v$.

**Possible moves** The possible actions of Scheduler are to schedule an active task or to idle the CPU. We model this by having, from all vertices $v \in V_{\Box}$ one transition labelled by some element from $\mathsf{active}(v)$, or by $\varepsilon$. The moves of TaskGen consist in selecting, for each task one possible *action* out of four: either (i) nothing ($\varepsilon$); or (ii) to finish the current job without submitting a new one ($fin$); or (iii) to submit a new job while the previous one is already finished ($sub$); or (iv) to submit a new job and kill the previous one, in the case of a soft task ($killANDsub$), which will incur a cost.

We consider the following example from [13].

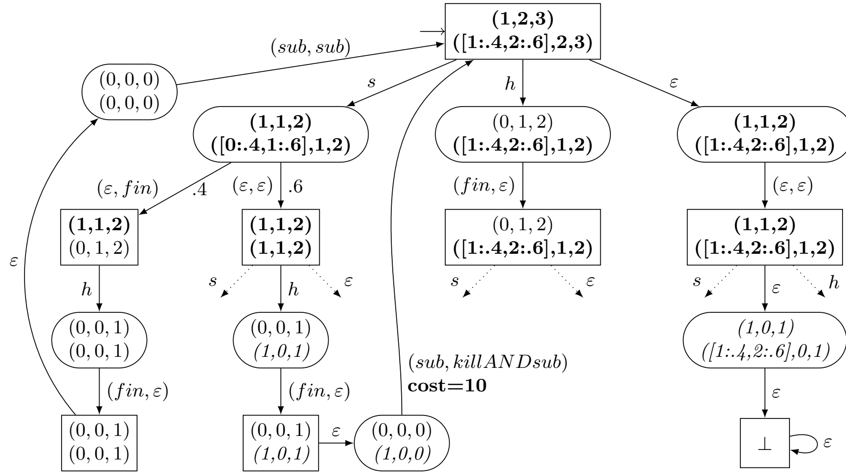


**Fig. 1.** MDP excerpt for Ex. 1. **Bold** tasks are active, those in *italics* have missed a deadline.

*Example 1.* Consider a system with one hard task $\tau_h = \langle \mathcal{C}_h, 2, \mathcal{A}_h \rangle$ s.t. $\mathcal{C}_h(1) = 1$ and $\mathcal{A}_h(3) = 1$; one soft task $\tau_s = \langle \mathcal{C}_s, 2, \mathcal{A}_s \rangle$ 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. $c(\tau_s) = 10$. Fig. 1 presents an excerpt of the MDP $\Gamma_\Upsilon$ built from the set of tasks $\tau = \{\tau_h, \tau_s\}$ of Example 1. A distribution $p$ with support $\{x_1, x_2, \ldots, x_n\}$ is denoted 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 $x$. Vertices from $V_{\Box}$ and $V_{\bigcirc}$ are depicted by rectangles and rounded rectangles respectively. Each vertex 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 $\tau_h$ consists in first scheduling $\tau_s$, then $\tau_h$. One then reaches the left-hand part of the graph from which $\Box$ can avoid $\bot$ whatever $\bigcirc$ does. Other safe strategies are possible: the first step of the algorithm in [13] is to compute all the *safe* nodes (i.e. those from which $\Box$ can ensure to avoid $\bot$), and then find an optimal one w.r.t to missed-deadline costs.

There are two optimal memoryless strategies, one in which Scheduler first chooses to execute $\tau_h$, then $\tau_s$; and another where $\tau_s$ is scheduled for 1 time

unit, and then preempted to let $\tau_h$ execute. Since the time difference between the arrival of two consecutive jobs of the soft task $\tau_s$ 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 over this time duration of 3, and hence the mean-cost is 2. There is another safe schedule that is not optimal which only grants $\tau_h$ is CPU access, and never schedules $\tau_s$, thus giving a mean-cost of $\frac{10}{3}$. □

**Expected mean-cost** Let us first associate a value, called the *mean-cost* $\mathsf{MC}(\pi)$ to all plays $\pi$ in an MDP $\Gamma = \langle V, E, L, (V_\Box, V_\bigcirc), A, \delta, \mathsf{cost}\rangle$. 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 $\mathsf{MC}$ is a measurable function. A strategy $\sigma_\Box$ is *optimal* for the mean-cost from some initial vertex $v_{\mathsf{init}} \in V_\Box$ if $\mathbb{E}_{v_{\mathsf{init}}}^{\Gamma[\sigma_\Box]}(\mathsf{MC}) = \inf_{\sigma'_\Box} \mathbb{E}_{v_{\mathsf{init}}}^{\Gamma[\sigma'_\Box]}(\mathsf{MC})$. Such *optimal* strategy always exists, and it is well-known that there is always one which is *memoryless*. Moreover, this problem can be solved in polynomial time through linear programming [11] or in practice using value iteration (as implemented, for example, in the tool Storm [9]). We denote by $\mathbb{E}_{v_{\mathsf{init}}}^{\Gamma}(\mathsf{MC})$ the optimal value $\inf_{\sigma_\Box} \mathbb{E}_{v_{\mathsf{init}}}^{\Gamma[\sigma_\Box]}(\mathsf{MC})$.

**Safety synthesis** Given an MDP $\Gamma = \langle V, E, L, (V_\Box, V_\bigcirc), A, \delta, \mathsf{cost}\rangle$, an initial vertex $v_{\mathsf{init}} \in V$, and a strategy $\sigma_\Box$, we define the set of possible *outcomes* in the Markov chain $\Gamma[\sigma_\Box]$ as the set of paths $v_{\mathsf{init}} = v_0 v_1 v_2 \dots$ in $\Gamma[\sigma_\Box]$ s.t., for all $i \geq 0$, there is non-null probability to go from $v_i$ to $v_{i+1}$ in $\Gamma[\sigma_\Box]$. Let $V_{\mathsf{Outs}_{\Gamma[\sigma_\Box]}(v_{\mathsf{init}})} \subseteq V$ denote the set of vertices visited in the set of possible outcomes $\mathsf{Outs}_{\Gamma[\sigma_\Box]}(v_{\mathsf{init}})$.

Given $\Gamma$ with vertices $V$, initial vertex $v_{\mathsf{init}} \in V$, and a set $V_{\mathsf{bad}} \subseteq V$ of *bad vertices*, the *safety synthesis problem* is to decide whether □ has a strategy $\sigma_\Box$ ensuring to visit the safe vertices only, i.e.: $V_{\mathsf{Outs}_{\Gamma[\sigma_\Box]}(v_{\mathsf{init}})} \cap V_{\mathsf{bad}} = \emptyset$. If this is the case, we call such a strategy *safe*. The safety synthesis problem is decidable in polynomial time for MDPs (see, e.g., safety games in [22]). Moreover, if a safe strategy exists, then there is a *memoryless* safe strategy. Henceforth, we will consider safe strategies that are memoryless only. We say that a vertex $v$ is safe iff □ has a safe strategy from $v$, and that an edge $e = (v, v') \in E \cap (V_\Box \times V_\bigcirc)$ is safe iff there is a safe strategy $\sigma_\Box$ s.t. $\sigma_\Box(v) = v'$. So, the *safe edges* $\mathsf{safe}(v)$ from some node $v$ correspond to the choices that □ can safely make from $v$. The set of safe edges exactly correspond to the set of safe actions that □ can make from $v$. Then, we let the *safe region* of $\Gamma$ be the MDP $\Gamma^{\text{safe}}$ obtained from $\Gamma$ by applying the following transformations: (i) remove from $\Gamma$ all *unsafe edges*; (ii) remove from $\Gamma$ all vertices and edges that are not reachable from $v_{\mathsf{init}}$.

**Most general safe scheduler** Consider a task system $\Upsilon$ that is schedulable for the hard tasks. Then, Scheduler has a winning strategy to avoid $\perp$ in $\Gamma_\Upsilon$. We say a non-deterministic strategy in $\Gamma_\Upsilon$ is the *most general safe scheduler* (MGS) for the hard tasks if from any vertex of Scheduler it allows all safe edges[4].

[4] The existence of a most general safe scheduler follows from the existence of a unique most general (a.k.a. maximally permissive) strategy for safety objectives [19].

## 3 Model-Based Learning

We now investigate the case of *model-based learning* of task systems. First, we consider the simpler case of task systems with only soft tasks. We show that those systems are always efficiently PAC learnable. Second, we consider learning task systems with both hard and soft tasks. In that case, we study two conditions for learnability. The first condition allows us to identify task systems that are safely PAC learnable, i.e. learnable while enforcing safety for the hard tasks. The second condition is stronger and allows us to identify task systems that are safely and *efficiently* PAC learnable.

**Learning setting** We consider a setting in which we are given the structure of a task system $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ to schedule. While the structure is known, the actual distributions that describe the behaviour of the tasks are unknown and need to be learnt to behave optimally or near optimally. The learning must be done only by observing the jobs that arrive along time. When the task system contains some hard tasks ($H \neq \emptyset$), all deadlines of such tasks must be enforced.

For learning the inter-arrival time distribution of a task, a *sample* corresponds to observing the time difference between the arrivals of two consecutive jobs of that task. For learning the computation time distribution, a sample corresponds to observing the CPU time a job of the task has been assigned up to completion. Thus if a job does not finish execution before its deadline, we do not obtain a valid sample for the computation time. Given a class of task systems, we say:

- the class is *probably approximately correct (PAC) learnable* if there is an algorithm $\mathbb{L}$ such that for all task systems $\Upsilon$ in this class, for all $\varepsilon, \gamma \in (0,1)$: given $\mathsf{struct}(\Upsilon)$, the algorithm $\mathbb{L}$ can execute the task system $\Upsilon$, and can compute $\Upsilon^M$ such that $\Upsilon \approx^{\varepsilon} \Upsilon^M$, with probability at least $1-\gamma$.
- the class is *safely PAC learnable* if it is PAC learnable, and $\mathbb{L}$ can ensure safety for the hard tasks while computing $\Upsilon^M$.
- the class is (safely) *efficiently PAC learnable* if it is (safely) PAC learnable, and there is a polynomial $q$ in the size of the task system, in $1/\varepsilon$, and in $1/\gamma$, s.t. $\mathbb{L}$ obtains enough samples to compute $\Upsilon^M$ in a time bounded by $q$.

Note that our notion of efficient PAC learning is stronger than the definition used in classical PAC learning terminology [23] since we take into account the time that is needed to get samples and not only the number of samples needed.

**Learning discrete finite distributions** To learn an unknown discrete distribution $p$ defined on a finite domain $\mathsf{Dom}(p)$, we collect i.i.d. samples from that distribution and infer a model of it. Formally, given a sequence $\mathbb{S} = (s_j)_{j\in J}$ of samples drawn i.i.d. from the distribution $p$, we denote by $p(\mathbb{S}) : \mathsf{Dom}(p) \to [0,1]$, the function that maps every element $a \in \mathsf{Dom}(p)$ to its relative frequency in $\mathbb{S}$. Using Hoeffding's inequality, it is easy to prove the following.

**Lemma 1.** *For all finite discrete distributions $p$ with $|\mathsf{Dom}(p)| = r$, for all $\varepsilon, \gamma \in (0,1)$ such that $\pi^p_{\min} > \varepsilon$, if $\mathbb{S}$ is a sequence of at least $r \cdot \lceil \frac{1}{2\varepsilon^2}(\ln 2r - \ln \gamma)\rceil$ i.i.d. samples drawn from $p$, then $p \sim^{\varepsilon} p(\mathbb{S})$ with probability at least $1-\gamma$.*

We say that we "PAC learn" a distribution $p$ if for all $\varepsilon, \gamma \in (0,1)$ such that $\pi^p_{\min} > \varepsilon$, by drawing a sequence $\mathbb{S}$ of i.i.d. samples from $p$, we have $p \sim^\varepsilon p(\mathbb{S})$ with probability at least $1-\gamma$. Given a task system $\Upsilon$, if we can learn the distributions corresponding to all the tasks in $\Upsilon$, and hence a model $\Upsilon^M$, such that each learnt distribution in $\Upsilon^M$ is structurally identical to its corresponding distribution in $\Upsilon$, the corresponding MDP are structurally identical.

**Efficient PAC learning** Let $\Upsilon = ((\tau_i)_{i\in I}, F, \emptyset)$ be a task system with soft tasks only, and let $\varepsilon, \gamma \in (0,1)$. We assume that for all distributions $p$ occurring in the models of the tasks in $\Upsilon$: $\pi^p_{\min} > \varepsilon$. To learn a model $\Upsilon^M$ which is $\varepsilon$-close to $\Upsilon$ with probability at least $1-\gamma$, we apply Lemma 1 in the following algorithm:

1. for all tasks $i = 1, 2, \dots \in F$, repeat the following learning phase:
   Always schedule task $\tau_i$ when a job of this task is active. Collect the samples $\mathbb{S}(\mathcal{A}_i)$ of $\mathcal{A}_i$ and $\mathbb{S}(\mathcal{C}_i)$ of $\mathcal{C}_i$ as observed. Collect enough samples to apply Lemma 1 and obtain the desired accuracy as fixed by $\varepsilon$ and $\gamma$.
2. the models of inter-arrival time distribution and computation time distribution for task $\tau_i$ are $p(\mathbb{S}(\mathcal{A}_i))$ and $p(\mathbb{S}(\mathcal{C}_i))$ respectively.

**Theorem 1.** *There is an algorithm s.t. for all task systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ with $H = \emptyset$, for all $\varepsilon, \gamma \in (0,1)$, it learns $\Upsilon^M$ s.t. $\Upsilon^M \approx^\varepsilon \Upsilon$ with probability at least $1-\gamma$ after executing $\Upsilon$ for $|F| \cdot \mathcal{A}_{\max} \cdot \mathbb{D} \cdot \lceil \frac{1}{2\varepsilon^2}(\ln 4\mathbb{D}|F| - \ln\gamma)\rceil$ steps.*

**Safe learning with hard tasks** We turn to task systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ with both hard and soft tasks. The learning algorithm must ensure that all the jobs of hard tasks meet their deadlines while learning the task distributions. The soft-task-only algorithm is clearly not valid for that more general case. Recall we have assumed schedulability of the task system for the hard tasks[5]. This is a necessary condition for safe learning but it is not a sufficient condition. Indeed, to apply Lemma 1, we need enough samples for all tasks $i \in H \cup F$.

First, we note that when executing any safe schedule for the hard tasks, we will observe enough samples for the hard tasks. Indeed, under a safe schedule for the hard tasks, any job of a hard task that enters the system will be executed to completion before its deadline. We then observe the value of the inter-arrival and computation times for all the jobs of hard tasks that enter the system. Unfortunately, this is not necessarily the case for soft tasks when they execute in the presence of hard tasks. Indeed, it is in general not possible to schedule all the jobs of soft tasks up to completion. We thus need stronger conditions in order to be able to learn the distributions of the soft tasks while ensuring safety.

**PAC guarantees for safe learning** Our condition to ensure safe PAC learnability relies on properties of the safe region $\Gamma^{\text{safe}}_\Upsilon$ in the MDP $\Gamma_\Upsilon$ associated to the task system $\Upsilon$. First, note that $\Gamma^{\text{safe}}_\Upsilon$ is guaranteed to be non-empty as the task system $\Upsilon$ is guaranteed to be schedulable for its hard tasks by hypothesis. Our condition will exploit the following property of its structure:

**Lemma 2.** *Let $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ be a task system and let $\Gamma^{safe}_\Upsilon$ be the safe region of its MDP. Then $\Gamma^{safe}_\Upsilon$ is a single maximal end-component (MEC).*

[5] Note that safety synthesis already identifies task systems that violate this condition.

**Good for sampling** The safe region $\varGamma_{\varUpsilon}^{\text{safe}}$ of the task system $\varUpsilon = ((\tau_i)_{i\in I}, F, H)$ is *good for sampling* if for all soft tasks $i \in F$, there exists a vertex $v_i \in \varGamma_{\varUpsilon}^{\text{safe}}$ such that: (i) a new job of task $i$ enters the system in $v_i$; and (ii) there exists a strategy $\sigma_i$ of Scheduler that is compatible with the set of safe schedules for the hard tasks so that from $v_i$, under schedule $\sigma_i$, the new job associated to task $\tau_i$ is guaranteed to reach completion before its deadline.

There is an algorithm that executes in polynomial time in the size of $\varGamma_{\varUpsilon}^{\text{safe}}$ and which decides if $\varGamma_{\varUpsilon}^{\text{safe}}$ is good for sampling. Also, remember that only the knowledge of the structure of the task system is needed to compute $\varGamma_{\varUpsilon}^{\text{safe}}$.

Given a task system $\varGamma_{\varUpsilon}^{\text{safe}}$ that is *good for sampling*, given any $\varepsilon, \gamma \in (0, 1)$, we safely learn a model $\varUpsilon^M$ which is $\varepsilon$-close to $\varUpsilon$ with probability at least $1 - \gamma$ (PAC guarantees) by applying the following algorithm:

1. Choose any safe strategy $\sigma_H$ for the hard tasks, and apply it until enough samples $(\mathbb{S}(\mathcal{A}_i), \mathbb{S}(\mathcal{C}_i))$ for each $i \in H$ have been collected according to Lemma 1. The models for tasks $i \in H$ are $p(\mathbb{S}(\mathcal{A}_i))$ and $p(\mathbb{S}(\mathcal{C}_i))$.
2. Then for each $i \in F$, apply the following phases:
   (a) from the current vertex $v$, schedule some task uniformly at random among the set of tasks that correspond to the safe edges in $\mathsf{safe}(v)$ up to reaching some $v_i$ (while choosing tasks that do not violate safety uniformly at random, we reach some $v_i$ with probability 1.[6] The existence of a $v_i$ is guaranteed by the hypothesis that $\varGamma_{\varUpsilon}^{\text{safe}}$ is good for sampling).
   (b) from $v_i$, apply the schedule $\sigma_i$ as defined by the second condition in the *good for sampling condition*. This way we are guaranteed to observe the computation time requested by the new job of task $i$ that entered the system in vertex $v_i$, no matter how TaskGen behaves. At the completion of this job of task $i$, we have collected a valid sample of task $i$.
   (c) go back to $(a)$ until enough samples $(\mathbb{S}(\mathcal{A}_i), \mathbb{S}(\mathcal{C}_i))$ have been collected for soft task $i$ according to Lemma 1.

**Theorem 2.** *There is an algorithm s.t. for all task systems $\varUpsilon = ((\tau_i)_{i\in I}, F, H)$ with a safe region $\varGamma_{\varUpsilon}^{safe}$ that is good for sampling, for all $\varepsilon, \gamma \in (0, 1)$, the algorithm learns a model $\varUpsilon^M$ such that $\varUpsilon^M \approx^{\varepsilon} \varUpsilon$ with probability at least $1 - \gamma$.*

In the algorithm above, to obtain one sample of a soft task, we need to reach a particular vertex $v_i$ from which we can safely schedule a new job for the task $i$ up to completion. As the underlying MDP $\varGamma_{\varUpsilon}^{\text{safe}}$ can be large (exponential in the description of the task system), we cannot bound by a polynomial the time needed to get the next sample in the learning algorithm. So, this algorithm does not guarantee efficient PAC learning. We develop in the next paragraph a stronger condition to guarantee efficient PAC learning.

**Good for efficient sampling** The safe region $\varGamma_{\varUpsilon}^{\text{safe}}$ of the task system $\varUpsilon = ((\tau_i)_{i\in I}, F, H)$ is *good for efficient sampling* if there exists $K \in \mathbb{N}$ which is bounded polynomially in the size of $\varUpsilon = ((\tau_i)_{i\in I}, F, H)$, and if, for all soft tasks $i \in F$ the two following conditions hold:

[6] This follows from the fact that there is a single MEC in the MDP by Lemma 2.

1. let $V_\Box^{\text{safe}}$ be the set of Scheduler vertices in $\Gamma_\Upsilon^{\text{safe}}$. There is a non-empty subset $\mathsf{Safe}_i \subseteq V_\Box^{\text{safe}}$ of vertices from which there is a strategy $\sigma_i$ for Scheduler to schedule safely the tasks $H \cup \{i\}$ (i.e. all hard tasks *and* the task $i$); and
2. for all $v \in V_\Box^{\text{safe}}, i \in F$, there is a uniform memoryless strategy $\sigma_{\Diamond\mathsf{Safe}_i}$ s.t.:
   (a) $\sigma_{\Diamond\mathsf{Safe}_i}$ is compatible with the safe strategies (for the hard tasks) of $\Gamma_\Upsilon^{\text{safe}}$;
   (b) when $\sigma_{\Diamond\mathsf{Safe}_i}$ is executed from any $v \in V_\Box^{\text{safe}}$, then the set $\mathsf{Safe}_i$ is reached within $K$ steps. By Lemma 2, since $\Gamma_\Upsilon^{\text{safe}}$ has a single MEC, we have that $\mathsf{Safe}_i$ is reachable from every $v \in V_\Box^{\text{safe}}$.

Here again, the condition can be efficiently decided: there is a polynomial-time algorithm in the size of $\Gamma_\Upsilon^{\text{safe}}$ that decides if $\Gamma_\Upsilon^{\text{safe}}$ is good for efficient sampling.

Given a task system $\Gamma_\Upsilon^{\text{safe}}$ that is *good for efficient sampling*, given $\varepsilon, \gamma \in (0,1)$, we safely and efficiently learn a model $\Upsilon^M$ which is $\varepsilon$-close of $\Upsilon$ with probability at least than $1-\gamma$ (efficient PAC guarantees) by applying:

1. Choose any safe strategy $\sigma_H$ for the hard tasks, and apply this strategy until enough samples $(\mathbb{S}(\mathcal{A}_i), \mathbb{S}(\mathcal{C}_i))$ for each $i \in H$ have been collected according to Lemma 1. The models for tasks $i \in H$ are $p(\mathbb{S}(\mathcal{A}_i))$ and $p(\mathbb{S}(\mathcal{C}_i))$.
2. Then for each $i \in F$, apply the following phase:
   (a) from the current vertex $v$, play $\sigma_{\Diamond\mathsf{Safe}_i}$ to reach the set $\mathsf{Safe}_i$.
   (b) from the current vertex in $\mathsf{Safe}_i$, apply the schedule $\sigma_i$ as defined above. This way we are guaranteed to observe the computation time requested by all the jobs of task $i$ that enter the system.
   (c) go to $(b)$ until enough samples $(\mathbb{S}(\mathcal{A}_i), \mathbb{S}(\mathcal{C}_i))$ are collected for task $i$ as per Lem. 1. The models for task $i$ are given by $p(\mathbb{S}(\mathcal{A}_i))$ and $p(\mathbb{S}(\mathcal{C}_i))$.

For a task system $\Upsilon$, let $T = \mathcal{A}_{\max} \cdot \mathbb{D} \cdot \lceil \frac{1}{2\varepsilon^2}(\ln 4\mathbb{D}|\Upsilon| - \ln \gamma) \rceil$. The properties of the learning algorithm above are used to prove the following theorem:

**Theorem 3.** *There is an algorithm s.t. for all systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ with safe region $\Gamma_\Upsilon^{safe}$ that is good for efficient sampling, for all $\varepsilon, \gamma \in (0,1)$, it learns $\Upsilon^M$ s.t. $\Upsilon^M \approx^\varepsilon \Upsilon$ with probability at least than $1-\gamma$ after scheduling $\Upsilon$ for $T + |F| \cdot (T+K)$ steps.*

**Using the learnt model** Given a system $\Upsilon$ of tasks, and parameters $\varepsilon, \gamma \in (0,1)$, once we have learnt a model $\Upsilon^M$ such that $\Upsilon^M \approx^\varepsilon \Upsilon$, we construct the MDP $\Gamma_{\Upsilon^M}^{\text{safe}}$. From $\Gamma_{\Upsilon^M}^{\text{safe}}$, we can compute an optimal scheduling strategy that minimises the expected mean-cost of missing deadlines of soft tasks. Such an algorithm is given in [13]. Then, we execute the actual task system $\Upsilon$ under schedule $\sigma$. However, since $\sigma$ has been computed using the model $\Upsilon^M$, it might not be optimal in the original, unknown taks system $\Upsilon$. Nevertheless, we can bound the difference between the optimal values obtained in $\Gamma_{\Upsilon^M}^{\text{safe}}$ and $\Gamma_\Upsilon^{\text{safe}}$.

The following lemma relates the model that is learnt with the approximate distribution that we have in the MDP corresponding to the learnt model. Given $\varepsilon \in (0,1)$, let $s = \min\{1, \pi_{\max}^\Upsilon + \varepsilon\}$ and $\eta = s^{2n} - (s-\varepsilon)^{2n}$, where $n = |\Upsilon|$.

**Lemma 3 (From [16]).** *Let $\Upsilon$ be a task system, let $\varepsilon, \gamma \in (0,1)$, let $\Upsilon^M$ be the learnt model such that $\Upsilon^M \approx^\varepsilon \Upsilon$ with probability at least $1-\gamma$. Then we have that $\Gamma_{\Upsilon^M} \approx^\eta \Gamma_\Upsilon$ with probability at least $1-\gamma$.*

A strategy $\sigma$ is said to be *(uniformly) expectation-optimal* if for all $v \in V_\square$, we have $\mathbb{E}_v^{\Gamma[\sigma]}(\mathsf{MC}) = \inf_\tau \mathbb{E}_v^{\Gamma[\tau]}(\mathsf{MC})$. The following Lemma captures the idea that some expectation-optimal strategies for MDPs whose transition functions have the same support as that of $\Gamma$ are 'robust'.

**Lemma 4 (From [7, Theorem 5]).** *Consider $\beta \in (0,1)$, and MDPs $\Gamma$ and $\Gamma'$ such that $\Gamma \approx^{\eta_\beta} \Gamma'$ with $\eta_\beta \leq \frac{\beta \cdot \pi_{\min}}{8|V_\square|}$, where $\pi_{\min}$ is the minimum probability appearing in $\Gamma$. For all memoryless deterministic expectation-optimal strategies $\sigma$ in $\Gamma'$, for all $v \in V_\square$, it holds that $\left|\mathbb{E}_v^{\Gamma[\sigma]}(\mathsf{MC}) - \inf_\tau \mathbb{E}_v^{\Gamma[\tau]}(\mathsf{MC})\right| \leq \beta$.*

The proof of the above lemma uses Thm. 6 in [21] and Thm. 5 in [7]. Using both Lemma 3 and Lemma 4, we obtain the following guarantees on the quality of the scheduler that our model-based learning algorithm outputs:

**Theorem 4.** *Given a task system $\Upsilon$ (with min probability $\pi_{\min}$) and $\beta \in (0,1)$. Let $\gamma, \varepsilon \in (0,1)$ be s.t. $\varepsilon \leq \frac{\beta \pi_{\min}}{8|V_\square| + \beta \pi_{\min}}$. Let $\Upsilon^M$ be s.t. $\Upsilon^M \approx^\varepsilon \Upsilon$ with probability at least $1-\gamma$, and let $\sigma$ be a memoryless deterministic expectation-optimal strategy of $\Gamma_{\Upsilon^M}$. Then, with probability at least $1-\gamma$, the expected mean-cost of playing $\sigma$ in $\Gamma_\Upsilon$ is s.t. for all $v \in V_\square$: $\left|\mathbb{E}_v^{\Gamma_\Upsilon[\sigma]}(\mathsf{MC}) - \inf_\tau \mathbb{E}_v^{\Gamma_\Upsilon[\tau]}(\mathsf{MC})\right| \leq \beta$.*

## 4 Monte Carlo Tree Search with Advice

When the model of the task system is known, or once it has been learned using techniques developed in Section 3, our goal is to compute a (near) optimal strategy while ensuring safe scheduling of hard-tasks with certainty.

The challenge is the sizes of the MDPs that are too large for exact model-checking techniques (see Sect. 5). To overcome this problem, we resort to a *receding horizon* framework [14], that bases its decisions on a finite-depth unfolding of the MDP from the current state. In particular, we advocate the use of *Monte Carlo Tree Search* (MCTS) algorithms [4], that are a popular method for sampling the finite-depth unfolding while avoiding an exponential dependency on the horizon. MCTS algorithms aim at discovering and exploring the "most relevant" parts of the unfolding, and they approximate the value of actions in intermediary nodes using a fixed number of trajectories obtained by simulations. The MCTS algorithm builds an exploration tree incrementally. At every step of the algorithm, the *selection phase* selects a path in the current tree, possibly extending it by adding a new node. It is followed by a *simulation phase*, that extends this trajectory further, until the fixed horizon is reached. Finally, a *back-propagation* phase updates the exploration tree based on this new trajectory. A reader looking for a more detailed introduction to MCTS is referred to [5].

MCTS has been successfully applied to large state-spaces. For example, it is an important building block of the AlphaGo algorithm [20] that has obtained super-human performances in the game of Go. Such level of performances cannot be obtained with the plain MCTS algorithm. In Go, the simulation and selection phases are guided by a board scoring function that has been learned using neural-networks techniques and self-play. For our scheduling problem, we also need a

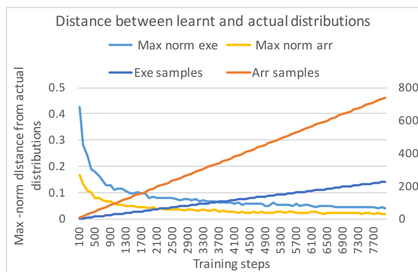


**Fig. 2.** Learning distributions for a system with 6 soft tasks.

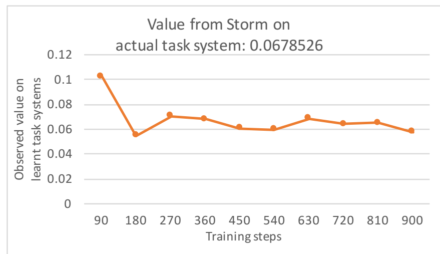


**Fig. 3.** Model-based learning for 1 hard, 2 soft tasks

solution to this guidance problem and, equally importantly, we must augment the MCTS algorithm in a way that *ensures* safe scheduling of hard tasks.

**Symbolic advice** In a recent previous work [5], we have introduced the notion of (symbolic) advice that provides a generic and formal solution to systematically incorporate domain knowledge in the MCTS algorithm. For our scheduling problem, we use selection advice that prunes parts of the MDP on-the-fly in order to ensure that only safe schedulers are explored. We have considered two possibilities. First, we consider the most general safe scheduler (MGS scheduler) as defined in page 7 to restrict the selection phase to safe scheduling decisions only. Second, we consider the earliest deadline first (EDF) scheduling strategy for hard tasks defined in page 4, that only allows soft tasks when there are no available hard tasks, and restricts to the hard tasks with the earliest deadline otherwise. EDF is guaranteed safe as the set of hard tasks is assumed schedulable. The MGS advice allows for maximal exploration as it leaves open all possible safe scheduling solutions, while the EDF advice can be applied on larger task systems as it does not require any precomputations. These advice are also applicable during the simulation phases.

## 5 Experimental Results

In this section, we first report experimental results on model-based learning and observe that the models are learnt efficiently with only a small number of samples. Our MCTS based algorithms can then be applied on the learnt models that are very close to the original ones.[7] We compare the performance of our MCTS-based algorithms with a state-of-the-art deep $Q$-learning implementation from OpenAI [10] on a set of benchmarks of task systems of various sizes. The experimental results show that our MCTS-based algorithms perform better in practice than safe reinforcement learning (RL)[3].

**Models with only soft tasks** In Figure 2, we show that the distributions of a task system with soft tasks can be learnt efficiently with a small number of samples, corroborating our theory in Section 3. This is not the case in general

[7] Here we do not learn to the point where our PAC guarantees hold. Rather, we are interested in how fast the learnt model converges to the real model in practice.

for arbitrary MDPs where in order to collect samples, one may need to reach some specific states of the MDP, and it may take a considerable amount of time to reach such states. However, in this case of systems with only soft tasks, the number of samples increases linearly with time. As a representative task system, we display the learning curve for a system with six soft tasks in Figure 2. Here "exe" and "arr" refer to the distributions of the computation times and the inter-arrival times respectively. The left $y$-axis is the max-norm distance between the probabilities in the actual distributions and the learnt distributions across all soft tasks. The $x$-axis is the number of time steps over which the system is executed. For learning the computation time distribution, the soft tasks are scheduled in a round robin manner. Once a job of a soft task is scheduled, it is executed until completion without being preempted. A sample for learning the computation time distribution of a soft task thus corresponds to a job of the task that is scheduled to execute until completion. Since the system has only soft tasks, a job can always be executed to finish its execution without safety being violated. On the other hand, the samples for learning the inter-arrival time distribution for each task correspond to all the jobs of the task that arrive in the system. Thus over a time duration, for each task, the number of samples collected for learning the inter-arrival time distribution is larger than the number of samples collected for learning the computation time distribution. The number of samples of both kinds increases linearly with time. The $y$-axis on the right corresponds to the number of samples collected over a duration of time when the system executes. The plot "Exe samples" corresponds to the number of samples collected per task for learning the computation time distributions. Since the tasks are executed in a round robin manner, the tasks have an equal number of samples for learning their computation time distributions. On the other hand, for learning inter-arrival time distributions, a task with larger inter-arrival time produces fewer samples than a task with smaller inter-arrival time. The plot "Arr samples" corresponds to the minimum of the number of jobs, over all the tasks, that arrived in the system. Each point in the graphs is obtained as a result of averaging over 50 simulations.

**Safe model-based learning** For safe model-based learning of systems with both hard and soft tasks, first, we verify that the task system satisfies the *good for efficient sampling* condition, and hence admits safe efficient PAC learning. We consider a small representative task system, and report the value of the optimal expected mean-cost strategy as computed by STORM on the learnt model as a function of the number of steps for which the system is executed (training steps). This converges quickly to the optimal expected value of the actual task system, roughly equal to 0.06 (see Fig 3). We also note that the expected value computed by STORM is not necessarily monotonic as it is computed on the learnt model and this model changes over time with the samples that it receives, and the expected value may also sometimes be smaller than the value on the actual model. The results show that this approach is effective in terms of the quality of learning and the number of samples required.

**MCTS** In the above approach, the main bottleneck towards scalability is the extraction of an optimal strategy from the learnt model using probabilistic model-checkers like Storm. This is because the underlying MDP grows exponentially with the number of tasks. Therefore we advocate the use of receding horizon techniques instead, that optimize the cost based on the next $h$ steps for some horizon $h$. In our examples, the unfoldings have approximately $2^h$ states, so we use MCTS to explore them in a scalable way.

**Deep Q-learning** One of the most successful model-free learning algorithm is the *Q-learning* algorithm, due to Watkins and Dayan [24]. It aims at learning (near) optimal strategies in a (partially unknown) MDP for the *discounted sum* objective. In our scheduling problem, we search for (near) optimal strategies for the mean-cost and *not* for the discounted sum, as we want to minimise the limit average of the cost of missing deadlines of soft tasks. However, if the discount factor is close to 1, both values coincide [21,17]. In our experiments, we use an implementation of deep $Q$-learning available in the OpenAI repository [10]. We make use of shielding [8,1,3], a technique that restricts actions in the learning process so that only those actions that are safe for the hard tasks can be used.

**Experimental setup for MCTS and deep Q-learning** We compare some variants of model-based learning augmented with MCTS and some variants of deep Q-learning in the context of scheduling. The first option is to set a very high penalty on missing the deadline of a hard task, and then to apply either MCTS or deep $Q$-learning. However, safety is not guaranteed in this case, and we report on whether a violation was observed or not. We call this variant unsafe MCTS and unsafe deep $Q$-learning respectively as a consequence. The second option is to enforce safety in MCTS and deep $Q$-learning by computing the most general safe scheduler for hard tasks, and then using the MGS advice for MCTS or the MGS shield for deep $Q$-learning. The third option is to use the earliest-deadline-first (EDF) scheme on hard tasks instead of MGS as an advice or a shield. Note that the second and the third options are required to ensure safety, and thus are applicable to systems that have at least one hard task, and hence are not applicable (NA) to systems with only soft tasks.

**Experimental Results** In the first column of Table 1, we describe the task systems that we consider. A description 2H, 5S refers to a task system with two hard tasks and five soft tasks, while 4S refers to a task system with four soft tasks and no hard tasks. The simple system refers to a 1H, 2S task system where all the arrival time distributions are Dirac. The output of Storm for the smaller task systems is given in the third column. We report sizes of the MDPs, computed with Storm whenever possible. Otherwise we report an approximation of the size of the state space obtained by taking the product of $(c_i + 1)(a_i + 1)$ over the set of tasks, where $c_i$ and $a_i$ are the greatest elements in the support of the distributions $\mathcal{C}_i$ and $\mathcal{A}_i$. Recall that the size of the state space is exponential in the number of tasks in the system. In the columns where safety is not guaranteed, $\infty$ denotes an observed violation (a missed deadline for a hard task).

For MCTS, at every step we explore 500 nodes of the unfolding of horizon 30, and the value of each node is initialized using 100 uniform simulations.

| Task | size | Storm output | MCTS unsafe | MCTS MGS | MCTS EDF | Deep-Q unsafe | Deep-Q MGS | Deep-Q EDF |
|---|---|---|---|---|---|---|---|---|
| 4S | $10^5$ | 0.38 | 0.52 | NA | NA | 0.56 | NA | NA |
| 5S | $10^6$ | T.O. | 0 | NA | NA | 0.13 | NA | NA |
| 10S | $10^{18}$ | T.O. | 0 | NA | NA | 0.96 | NA | NA |
| simple | $10^2$ | 0 | 0.72 | 0 | 0 | 1.08 | 0.1 | 0 |
| 1H, 2S | $10^4$ | 0.07 | 0.67 | 0.14 | 0.28 | 0.24 | 0.11 | 0.22 |
| 1H, 3S | $10^5$ | 0.28 | 1.13 | 0.45 | 0.49 | $\infty$ | 0.47 | 0.47 |
| 2H, 1S | $10^4$ | 0 | 0.92 | 0 | 0.2 | $\infty$ | 0.02 | 0.3 |
| 2H, 5S | $10^{10}$ | T.O. | 3.44 | 1.93 | 2.14 | $\infty$ | 2.39 | 2.48 |
| 3H, 6S | $10^{14}$ | T.O. | 4.17 | 2.88 | 2.97 | $\infty$ | 3.42 | 3.47 |
| 2H, 10S | $10^{22}$ | T.O. | 0.3 | 0.03 | 0.03 | $\infty$ | 1.42 | 1.6 |
| 4H, 12S | $10^{30}$ | T.O. | 2.1 | 1.2 | 1.3 | $\infty$ | 2.68 | 2.87 |

**Table 1.** Comparison of MCTS and reinforcement learning.

This computation takes 1-4 minutes in our Python implementation for different benchmarks, running on a standard laptop. It is reasonable to believe that a substantial speedup could be obtained with well-optimised code and parallelism. For deep $Q$-learning, we train each task system for 10000 steps. The implementation of deep-Q learning in the OepnAI respository uses the Adam optimizer [15]. The size of the replay buffer is set to 2000. The learning rate used is $10^{-3}$. The probability $\varepsilon$ of taking a random action is initially set to 1. This parameter reduces over the training steps, and becomes equal to 0.02 at the end of the training. The network used is a multi-layer perceptron which, by default, uses two fully connected hidden layers, each with 64 nodes. Since we are interested in mean-cost objective, the discount factor $\gamma$ is set to 1. We observed that reducing the value of $\gamma$ leads to poorer results. The values reported for both MCTS and deep $Q$-learning are obtained as an average cost over 600 steps.

**Conclusions** While deep Q-learning provides good results for small task systems with 3-4 tasks with several thousands of states, this method does not perform well for the benchmarks with large number of tasks. We trained the task system with 10 soft tasks with deep Q-learning for several million steps, but the state space was found to be too large to learn a good strategy, and the resulting output produced a cost that is much higher than that observed with MCTS.

Overall, our experimental results show that MCTS consistently provides better results, in particular when the task systems are large, with huge state spaces. This can be explained by the fact that MCTS optimizes locally using information about multiple possible “futures” while deep Q-learning rather optimizes globally using information about the uniquely observed trace. We observe that the performance of MCTS with EDF advice is only slightly worse than MCTS with MGS advice. EDF guarantees safety and does not require computing the most general safe strategy, therefore it forms a good heuristic for systems with many hard tasks, where MGS computation becomes too expensive.

In future work, we consider using Deep-$Q$ learning in either a selection advice for MCTS or as a complement to simulations when evaluating new states.

## References


1. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI. pp. 2669–2678. AAAI Press (2018)
2. Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: Deep reinforcement learning: A brief survey. IEEE Signal Process. Mag. **34**(6), 26–38 (2017)
3. Avni, G., Bloem, R., Chatterjee, K., Henzinger, T.A., Könighofer, B., Pranger, S.: Run-time optimization for learned controllers through quantitative games. In: CAV. pp. 630–649 (2019)
4. Browne, C., Powley, E.J., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Liebana, D.P., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Trans. on Computational Intelligence and AI in Games **4**(1), 1–43 (2012). https://doi.org/10.1109/TCIAIG.2012.2186810
5. Busatto-Gaston, D., Chakraborty, D., Raskin, J.: Monte Carlo Tree Search Guided by Symbolic Advice for MDPs. In: CONCUR. pp. 40:1–40:24 (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.40
6. Buttazzo, G.C.: Hard real-time computing systems: predictable scheduling algorithms and applications, vol. 24. Springer Science & Business Media (2011)
7. Chatterjee, K.: Robustness of structurally equivalent concurrent parity games. In: FOSSACS. pp. 270–285 (2012)
8. Chatterjee, K., Novotný, P., Pérez, G.A., Raskin, J.F., Zikelic, D.: Optimizing expectation with guarantees in pomdps. In: AAAI. pp. 3725–3732 (2017)
9. Dehnert, C., Junges, S., Katoen, J., Volk, M.: A storm is coming: A modern probabilistic model checker. In: CAV (2017)
10. Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., Zhokhov, P.: Openai baselines. `https://github.com/openai/baselines` (2017)
11. Filar, J., Vrieze, K.: Competitive Markov decision processes. Springer (1997)
12. Fu, J., Topcu, U.: Probably approximately correct MDP learning and control with temporal logic constraints. In: Fox, D., Kavraki, L.E., Kurniawati, H. (eds.) Robotics: Science and Systems X, University of California, Berkeley, USA, July 12-16, 2014 (2014). https://doi.org/10.15607/RSS.2014.X.039, `http://www.roboticsproceedings.org/rss10/p39.html`
13. Geeraerts, G., Guha, S., Raskin, J.F.: Safe and optimal scheduling for hard and soft tasks. In: FSTTCS. LIPIcs, vol. 122, pp. 36:1–36:22 (2018)
14. Kearns, M.J., Mansour, Y., Ng, A.Y.: A sparse sampling algorithm for near-optimal planning in large Markov decision processes. Machine Learning **49**(2-3), 193–208 (2002). https://doi.org/10.1023/A:1017932429737
15. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: ICLR (2015)
16. Kretínský, J., Pérez, G.A., Raskin, J.F.: Learning-based mean-payoff optimization in an unknown MDP under omega-regular constraints. In: CONCUR. LIPIcs (2018)
17. Mertens, J.F., Neyman, A.: Stochastic games. International Journal of Game Theory **10**(2), 53–66 (1981)
18. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (Feb 2015)

19. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. on Cont. and Opt. **25**(1), 206–230 (1987)
20. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T.P., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016). https://doi.org/10.1038/nature16961
21. Solan, E.: Continuity of the value of competitive markov decision processes. Journal of Theoretical Probability **16**, 831–845 (2003)
22. Thomas, W.: On the synthesis of strategies in infinite games. In: STACS. pp. 1–13 (1995)
23. Valiant, L.G.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)
24. Watkins, C.J.C.H., Dayan, P.: Technical note Q-learning. Machine Learning **8**, 279–292 (1992)

# Appendix

## A Proof of Theorem 1

*There is a learning algorithm such that for all task systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ with $H = \emptyset$, for all $\varepsilon, \gamma \in (0,1)$, the algorithm learns a model $\Upsilon^M$ such that $\Upsilon^M \approx^{\varepsilon} \Upsilon$ with probability at least $1-\gamma$ after executing $\Upsilon$ for $|F| \cdot \mathcal{A}_{\max} \cdot \mathbb{D} \cdot \lceil \frac{1}{2\varepsilon^2}(\ln 4\mathbb{D}|F| - \ln\gamma)\rceil$ steps.*

*Proof.* Using Lemma 1, given $\varepsilon, \gamma' \in (0,1)$, for every distribution $p$ of the task system, a sequence $\mathbb{S}$ of $\mathbb{D} \cdot \lceil \frac{1}{2\varepsilon^2}(\ln 2\mathbb{D} - \ln\gamma')\rceil$ i.i.d. samples suffices to have $p(\mathbb{S}) \sim^{\varepsilon} p$ with probability at least $1-\gamma'$. Since in the task system $\Upsilon$, there are $2|F|$ distributions, with probability at least $1-2|F|\gamma'$, we have that the learnt model $\Upsilon^M \approx^{\varepsilon} \Upsilon$. Thus for $\gamma' = \frac{\gamma}{2|F|}$, and using $2\exp(-2m\varepsilon^2) \leq \frac{\gamma}{2|F|\mathbb{D}}$, we have that for each distribution, a sequence of $\mathbb{D} \cdot \lceil \frac{1}{2\varepsilon^2}(\ln 4\mathbb{D}|F| - \ln\gamma)\rceil$ samples suffices so that $\Upsilon^M \approx^{\varepsilon} \Upsilon$ with probability at least $1-\gamma$.

Since samples for computation time distribution and inter-arrival time distribution for each soft task can be collected simultaneously, and observing each sample takes a maximum of $\mathcal{A}_{\max}$ time steps, and we collect samples for each soft task by scheduling one soft task after another, the result follows. □

## B Proof of Lemma 2

*Let $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ be a task system and let $\Gamma_{\Upsilon}^{safe}$ be the safe region of its MDP. Then $\Gamma_{\Upsilon}^{safe}$ is a single MEC.*

*Proof.* We first assume that the task system $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ is schedulable. Otherwise, $\Gamma_{\Upsilon}^{\mathrm{safe}}$ is empty and the Lemma is trivially true. Let $V$ and $E$ be the set of vertices and the set of edges of $\Gamma_{\Upsilon}^{\mathrm{safe}}$ respectively. First, observe that, since we want to prove that the whole MDP $\Gamma_{\Upsilon}^{\mathrm{safe}}$ corresponds to an MEC, we only need to show that its underlying graph $(V, E)$ is strongly connected. Indeed, since $(V, E)$ contains all vertices and edges from $\Gamma_{\Upsilon}^{\mathrm{safe}}$, it is necessarily maximal, and all choices of actions from any vertex will always lead to a vertex in $V$.

In order to show the strongly connected property, we fix a vertex $v \in V$, and show that there exists a path in $\Gamma_{\Upsilon}^{\mathrm{safe}}$ from $v$ to $v_{\mathsf{init}}$. Since all vertices in $V$ are, by construction of $\Gamma_{\Upsilon}^{\mathrm{safe}}$, reachable from the initial vertex $v_{\mathsf{init}}$, this entails that all vertices $v'$ are also reachable from $v$, hence, the graph is strongly connected.

Let us first assume that $v \in V_{\Box}$, i.e., $v$ is a vertex where Scheduler has to take a decision. Let $v_{\mathsf{init}} = v_0, v'_0, v_1, v'_1, \cdots, v'_{n-1}, v_n = v$ be the path $\pi$ leading to $v$, where all vertices $v_j$ belong to Scheduler, and all $v'_j$ are are vertices that belong to TaskGen.

Then, from path $\pi$, we extract, for all tasks $\tau_i$ the sequence of *actual inter-arrival times* $\sigma_i = t^i(1), t^i(2), \ldots, t^i(k_i)$ defined as follows: for all $1 \leq j \leq k_i$, $t^i(j) \in \mathsf{Supp}(\mathcal{A}_i)$ is the time elapsed (in CPU ticks) between the arrival of the $j-1$th job the $j$th job of task $i$ along $\pi$ (assuming the initial release

occurring in the initial state $v_{\mathsf{init}}$ is the 0-th release). In other words, letting $T^i(j) = \sum_{k=1}^{j} t^i(k)$, the $j$th job of $\tau_i$ is released along $\pi$ on the transition between $v'_{T^i(j-1)}$ and $v_{T^i(j)}$. Observe thus that all tasks $i \in [n]$ are in the same state in vertex $v_{\mathsf{init}}$ and in vertex $v_{T^i(j)}$, i.e. the time to the deadline, and the probability distributions on the next arrival and computation times are the same in $v_{\mathsf{init}}$ and $v_{T^i(j)}$. However, the vertices $v_{T^i(j)}$ can be different for all the different tasks, since they depend on the sequence of job releases of $\tau_i$ along $\pi$. Nevertheless, we claim that $\pi$ can be extended, by repeating the sequence of arrivals of all the tasks along $\pi$, in order to reach a vertex where all tasks have just submitted a job (i.e. $v_{\mathsf{init}}$). To this aim, we first extend, for all tasks $i \in [i]$, $\sigma_i$ into $\sigma'_i = \sigma_i, t^i(k_i + 1)$, where $t^i(k_i + 1) \in \mathsf{Supp}(\mathcal{A}_i)$ ensures that the $k_i + 1$ arrival of a $\tau_i$ occurs *after* $v$.

For all $i \in [n]$, let $\Delta_i$ denote $\sum_{j=1}^{k_i+1} t^i(j)$, i.e. $\Delta_i$ is the total number of CPU ticks needed to reach the first state after $v$ where task $i$ has just submitted a job (following the sequence of arrival $\sigma'_i$ defined above). Further, let $\Delta = \mathrm{lcm}(\Delta_i)_{i\in[n]}$. Now, let $\pi'$ be a path in $\Gamma_{\Upsilon}^{\mathrm{safe}}$ that respects the following properties:

1. $\pi$ is a prefix of $\pi'$;
2. $\pi'$ has a length of $\Delta$ CPU ticks;
3. $\pi'$ ends in a $\Box$ vertex $v'$; and
4. for all tasks $i \in [n]$: $\tau_i$ submits a job at time $t$ along $\pi'$ iff it submits a job at time $t \mod \Delta_i$ along $\pi$.

Observe that, in the definition of $\pi'$, we do not constrain the decisions of Scheduler after the prefix $\pi$. First, let us explain why such a path exists. Observe that the sequence of task arrival times is legal, since it consists, for all tasks $i$, in repeating $\Delta/\Delta_i$ times the sequence $\sigma'_i$ of inter-arrival times which is legal since it is extracted from path $\pi$ (remember that nothing that Scheduler player does can restrict the times at which TaskGen introduces new jobs in the system). Then, since $\Upsilon$ is schedulable, we have the guarantee that all $\Box$ vertices in $\Gamma_{\Upsilon}^{\mathrm{safe}}$ have at least one outgoing edge. This is sufficient to ensure that $\pi'$ indeed exists. Finally, we observe $\pi'$ visits $v$ (since $\pi$ is a prefix of $\pi'$), and that the last vertex $v'$ of $\pi'$ is a $\Box$ vertex obtained just after *all tasks* have submitted a job, by construction. Thus $v' = v_{\mathsf{init}}$, and we conclude that, from all $v \in V_{\Box}$ which is reachable from $v_{\mathsf{init}}$, one can find a path in $\Gamma_{\Upsilon}^{\mathrm{safe}}$ that leads back to $v_{\mathsf{init}}$.

This reasoning can be extended to account for the nodes $v \in V_{\bigcirc}$: one can simply select any successor $\overline{v} \in V_{\Box}$ of $v$, and apply the above reasoning from $\overline{v}$ to find a path going back to $v_{\mathsf{init}}$. $\Box$

## C Proof of Theorem 2

*There is a learning algorithm such that for all task systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ with a safe region $\Gamma_{\Upsilon}^{safe}$ that is good for sampling, for all $\varepsilon, \gamma \in (0, 1)$, the algorithm learns a model $\Upsilon^M$ such that $\Upsilon^M \approx^{\varepsilon} \Upsilon$ with probability at least $1 - \gamma$.*

*Proof.* For the hard tasks, as mentioned above, we can learn the distributions by applying the safe strategy $\sigma_H$ to collect enough samples $(\mathbb{S}(\mathcal{A}_i), \mathbb{S}(\mathcal{C}_i))$ for each $i \in H$.

We assume an order on the set of soft tasks. First for all $\tau_i$ for $i \in F$, since $\Gamma_{\Upsilon}^{\text{safe}}$ is good for sampling, we note that the set $V_i$ of vertices $v_i$ (as defined in the definition of good for sampling condition) is non-empty. Recall from Lemma 2 that $\Gamma_{\Upsilon}^{\text{safe}}$ has a single MEC. Thus from every vertex of $\Gamma_{\Upsilon}^{\text{safe}}$, Scheduler by playing uniformly at random reaches some $v_i \in V_i$ with probability 1, and hence can visit the vertices of $V_i$ infinitely often with probability 1. Now given $\varepsilon$ and $\gamma$, using Theorem 1, we can compute an $m$, the number of samples corresponding to each distribution required for safe PAC learning of the task system. Since by playing uniformly at random, Scheduler has a strategy to visit the vertices of $V_i$ infinitely often with probability 1, it is thus possible to visit these vertices at least $m$ times with arbitrarily high probability.

Also after we safely PAC learn the distributions for task $\tau_i$, since there is a single MEC in $\Gamma_{\Upsilon}^{\text{safe}}$, there exists a uniform memoryless strategy to visit a vertex $v_{i+1}$ corresponding to task $\tau_{i+1}$ with probability 1. Hence the result. □