Probabilistic Hyperproperties with Rewards*

Oyendrila Dobe[1]**, Lukas Wilke[2]**, Erika Ábrahám[2], Ezio Bartocci[3], and Borzoo Bonakdarpour[1]

[1] Michigan State University, East Lansing, MI, USA,
[2] RWTH Aachen University, Aachen, Germany,
[3] Technische Universität Wien, Vienna, Austria.

Abstract. *Probabilistic hyperproperties* describe system properties that are concerned with the probability relation between different system executions. Likewise, it is desirable to relate performance metrics (e.g., energy, execution time, etc) between multiple runs. This paper introduces the notion of *rewards* to the temporal logic HyperPCTL by extending the syntax and semantics of the logic to express the accumulated reward relation among different computations. We demonstrate the application of the extended logic in expressing side-channel timing countermeasures, efficiency in probabilistic conformance, path planning in robotics applications, and recovery time in distributed self-stabilizing systems. We also propose a model checking algorithm for verifying Markov Decision Processes against HyperPCTL with rewards and report experimental results.

1 Introduction

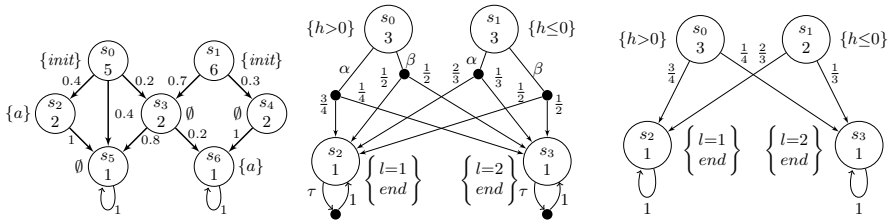
Stochastic phenomena appear in many systems such as those that interact with the physical environment (e.g., due to environmental uncertainties, thermal fluctuations, random message loss, and processor failure). Traditionally, the specification of systems that deal with uncertainties are expressed in some form of probabilistic temporal logic such as PCTL and PCTL* [5]. These logics can express the properties of *single* probabilistic computation trees. The temporal logic HyperPCTL [2] generalizes PCTL to express *probabilistic hyperproperties* by allowing quantification over multiple computation trees and expressing the probability relation among them. For instance, consider the Markov Decision Process (MDP) in Fig. 1b. The HyperPCTL formula

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\ \Big((h>0)_{\hat{s}} \wedge (h\leq 0)_{\hat{s}'}\Big) \Rightarrow \Big(\mathbb{P}\Diamond(l=1)_{\hat{s}} = \mathbb{P}\Diamond(l=1)_{\hat{s}'}\Big)$$

requires that the probability of reaching a state with proposition $l=1$ from any pair of states $\hat{s}$ and $\hat{s}'$ labeled by $h>0$ and $h\leq 0$ respectively, should be equal for the Discrete Time Markov Chain (DTMC) induced by any scheduler $\hat{\sigma}$.

* This research was partially supported by the United States NSF SaTC Award 2100989, WWTF ICT19-018 grant ProbInG and the DFG Research and Training Group UnRAVeL.

** First co-authors.



(a) A DTMCR. (b) An MDPR. (c) An induced DTMCR.

Fig. 1: Example Markov models.

In addition to the probability relation between certain events and computations, it is natural to analyze the average behavior of Markov models as well as the interrelation of average behaviors in different executions. For example:

- *Service-level agreements* (e.g., average system response time and uptime) are generally concerned with average performance metrics of a system among a set of executions. This is, of course, a system-wide performance requirement rather than the property of individual executions.
- *Side-channel timing* leaks can potentially reveal sensitive information through execution time of a function call. The execution time can be captured as a reward model where each instruction is associated with a cost and the probabilistic hyperproperty expresses that every pair of executions should exhibit the same expected execution cost.
- *Distributed algorithms* often use randomization to break symmetry in order to tackle impossibility results. Although one can reason about the expected performance of a randomized distributed algorithm by the traditional reward models, from a design perspective, it is desirable to determine and mitigate states from where convergence to the objective of the algorithm takes much longer than others.

These examples clearly motivate the need to somehow augment probabilistic hyperproperties with reward constraints.

With this motivation, our first contribution in this paper is to make the connection between reward models and probabilistic hyperproperties. In the context of a hyperproperty, analogous to the probability relation between multiple executions in a HyperPCTL formula, a reward mechanism should be able to express the expected reward relation along different quantified computation trees. To this end, we extend the syntax and semantics of HyperPCTL by allowing arithmetic functions over expected rewards and comparing them over multiple executions. For instance, for the MDP in Fig. 1b one may express whether there exist two schedulers such that starting from any two states, labeled with $h{>}0$ and $h{\leq}0$, resp., the expected reward of reaching an *end*-labeled state is the same using the following property:

$$\exists\hat{\sigma}_1.\exists\hat{\sigma}_2.\forall\hat{s}(\hat{\sigma}_1).\forall\hat{s}'(\hat{\sigma}_2).\Big((h{>}0)_{\hat{s}} \wedge (h{\leq}0)_{\hat{s}'}\Big) \rightarrow \Big(\mathcal{R}_{\hat{s}}(\Diamond\, end_{\hat{s}}) = \mathcal{R}_{\hat{s}'}(\Diamond\, end_{\hat{s}'})\Big)$$

In the MDP in Fig. 1b, if we instantiate $\hat{s}$ with s_0, and choose the action α, we collect a reward of $(3 + \frac{3}{4} \times 1 + \frac{1}{4} \times 1) = 4$, on reaching s_2 and s_3 with label *end*. Similarly, if we instantiate $\hat{s}'$ with s_1, and choose the action α, we collect a reward of $(3 + \frac{2}{3} \times 1 + \frac{1}{3} \times 1) = 4$, on reaching s_2 and s_3 with label *end*. Hence, we can prove the existence of schedulers that satisfy the above property in the MDP in Fig. 1b. On a closer look, no matter which action we choose at s_0 and s_1, the property is always satisfied. Also, if we instantiate $\hat{s}$ and $\hat{s}'$ with any other states different from s_0 resp. s_1, the property is vacuously true. On the contrary, if we replace the equality of rewards with inequality then the property is false as there are no such schedulers. Besides comparing reward values, our HyperPCTL extension offers further expressive power to e.g. measure accumulated rewards in an execution until an observable property, say termination, gets satisfied in another one.

Our second contribution in this paper is an algorithm for model checking HyperPCTL formulas with rewards for MDPs. Since the general verification problem is known to be undecidable [2], we focus on memoryless non-probabilistic schedulers which yields a decidable problem, for which we propose a model checking algorithm based on logical problem encoding and SMT solving. We have implemented a prototype of our method and analyzed it experimentally on three case studies: (1) side-channel timing attacks, (2) probabilistic performance conformance, and (3) randomized path planning for multi-agent robotics applications.

Organization. In Section 2, we present preliminary concepts. We introduce our proposed extension of HyperPCTL with rewards in Section 3 and discuss its applications in Section 4. We present our model checking algorithm and associated experimental results in Sections 5 and 6, respectively. Related work is discussed in Section 7. Finally, we conclude in Section 8.

2 Preliminaries

By $\mathbb{R}$ ($\mathbb{R}_{\geq 0}$) we denote the real (non-negative real) numbers, and by $\mathbb{N}$ the natural numbers including 0. For any domain D and any $v = (v_0, \ldots, v_{n-1}) \in D^n$, we define $v[i] = v_i$ for $i \in \{0, \ldots, n-1\}$. The concepts below have been adapted from [5] and extended to work for hyperlogics.

2.1 Discrete-time Markov models with rewards

When defining costs or rewards for Markov models, we can assign rewards to states or transitions. In this work we limit to the assignment of *non-negative* rewards to *states* and support multi-dimensional reward *vectors*.

Definition 1. *A* Discrete Time Markov Chain with (k-ary) rewards (DTMCR) *is a tuple* $\mathcal{D} = (S, P, \mathsf{AP}, L, rew)$ *with (1) a non-empty set of states* S*, (2) a transition function* $P : S \times S \to [0, 1] \subseteq \mathbb{R}$ *with* $\sum_{s' \in S} P(s, s') = 1$ *for all* $s \in S$*, (3) a finite set of atomic propositions* AP*, (4) a labeling function* $L : S \to 2^{\mathsf{AP}}$ *and (5) a reward function* rew*:* $S \to \mathbb{R}^k_{\geq 0}$.

Fig. 1a shows an example DTMCR with unary rewards. Assume a DTMCR $\mathcal{D} = (S, P, \mathsf{AP}, L, rew)$. An *infinite path* is a sequence of states $\pi = s_0 s_1 \ldots \in S^\omega$ with $P(s_i, s_{i+1}) > 0$ for all $i \in \mathbb{N}$. A non-empty prefix of an infinite path is a *finite path* $\pi = s_0 \ldots s_{n-1} \in S^+$ of length $|\pi| = n \in \mathbb{N} \setminus \{0\}$. Let $Paths_s^{\mathcal{D}}$ ($fPaths_s^{\mathcal{D}}$) be the set of all infinite (finite) paths starting in $s \in S$. A state $t \in S$ is *reachable* from $s \in S$ if there exists a path in $fPaths_s^{\mathcal{D}}$ ending in t. A state $s \in S$ is *absorbing* iff $P(s, s) = 1$.

For a finite path $\pi \in fPaths_s^{\mathcal{D}}$, we define its *cylinder set* $Cyl^{\mathcal{D}}(\pi)$ as the set of all infinite paths with π as a prefix. The probability of the cylinder set of $\pi \in fPaths_s^{\mathcal{D}}$ is defined as $\Pr_s^{\mathcal{D}}(Cyl^{\mathcal{D}}(\pi)) = \Pi_{i=0}^{|\pi|-1} P(s_i, s_{i+1})$. For sets $R \subseteq fPaths_s^{\mathcal{D}}$ we have $\Pr_s^{\mathcal{D}}(R) = \sum_{\pi \in R'} \Pr_s^{\mathcal{D}}(\pi)$, where R' contains all finite paths from R that have no extensions in R. These notions induce for each $s \in S$ the *probability space*,

$$\Big(Paths_s^{\mathcal{D}}, \big\{ \bigcup_{\pi \in R} Cyl^{\mathcal{D}}(\pi) \mid R \subseteq fPaths_s^{\mathcal{D}} \big\}, \Pr_s^{\mathcal{D}} \Big).$$

Note that the cylinder sets of two finite paths starting in the same state are either disjoint or one is contained in the other.

For a reward function $rew\colon S \to \mathbb{R}_{\geq 0}^k$ and $i \in \{0, \ldots, k-1\}$ we define $rew_i\colon S \to \mathbb{R}_{\geq 0}$ to assign the *ith state reward* $rew_i(s) = rew(s)[i]$ to all $s \in S$. The *ith cumulative reward* for a finite path, $\pi = s_0 s_1 \ldots s_{n-1}$ is defined as $rew_i(\pi) = \sum_{j=0}^{n-1} rew_i(s_j)$. Note that non-negative rewards assure monotonic increase of cumulative rewards with path extensions.

To argue about simultaneous runs across two DTMCRs, we define their parallel composition.

Definition 2. *Assume two DTMCRs $\mathcal{D}_i = (S_i, P_i, \mathsf{AP}_i, L_i, rew_i)$ with k_i-ary rewards, $i \in \{1, 2\}$. We define the* parallel composition $\mathcal{D}_1 \times \mathcal{D}_2 = (S_1 \times S_2, P, \mathsf{AP}_1 \cup \mathsf{AP}_2, L, rew)$ *with (k_1+k_2)-ary rewards, such that for all (s_1, s_2), $(s_1', s_2') \in S \times S$: (1) $P\big((s_1, s_2), (s_1', s_2')\big) = P_1(s_1, s_1') \cdot P_2(s_2, s_2')$, (2) $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$ and (3) $rew((s_1, s_2)) = (rew_1(s_1), rew_2(s_2))$.*

Next, we extend the probabilistic nature of DTMCRs with non-determinism.

Definition 3. *A* Markov Decision Process with k-ary rewards (MDPR) *is a tuple $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$ with (1) a non-empty set of states S, (2) a non-empty finite set of actions Act, (3) a transition function $P : S \times Act \times S \to [0, 1] \subseteq \mathbb{R}$ such that for each $s \in S$ we have $\sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\}$. For all $\alpha \in Act$, there is at least one action that can be chosen in each state, such that $\alpha \in Act(s) = \{\alpha \in Act \mid \sum_{s' \in S} P(s, \alpha, s') = 1\}$ and for $\alpha \in Act \setminus Act(s), \sum_{s' \in S} P(s, \alpha, s') = 0$, (4) a finite set of atomic propositions AP, (5) a labeling function $L : S \to 2^{\mathsf{AP}}$, and (6) a reward function $rew : S \to \mathbb{R}_{\geq 0}^k$.*

Fig.1b shows an example MDPR. In each state, for the next execution step, any of the enabled actions can be chosen non-deterministically. *Schedulers* are used to eliminate this non-determinism.

Definition 4. *A* scheduler *for an MDPR $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$ is a tuple $\sigma = (Q, act, mode, init)$ with (1) a countable set of modes Q, (2) a function $act : Q \times S \times Act \to [0,1] \subseteq \mathbb{R}$ such that for every $s \in S$ and $q \in Q$,*

$$\textstyle\sum_{\alpha \in Act(s)} act(q,s,\alpha) = 1 \quad \text{and} \sum_{\alpha \in Act \setminus Act(s)} act(q,s,\alpha) = 0 \ ,$$

(3) a mode transition function $mode : Q \times S \to Q$ and (4) $init : S \to Q$ assigning to each state of $\mathcal{M}$ a starting mode.

Let $\Sigma^{\mathcal{M}}$ be the set of all schedulers for $\mathcal{M}$. A scheduler is *finite-memory* if Q is finite, *memoryless* if $|Q| = 1$, and *non-probabilistic* if $act(q,s,\alpha) \in \{0,1\}$ for all $q \in Q$, $s \in S$ and $\alpha \in Act$.

Definition 5. *Assume an MDPR $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$ with k-ary rewards and a scheduler $\sigma = (Q, act, mode, init)$ for $\mathcal{M}$. Then $\mathcal{M}$ and σ* induce *the DTMCR with k-ary rewards $\mathcal{M}^\sigma = (S^\sigma, P^\sigma, \mathsf{AP}, L^\sigma, rew^\sigma)$, where $S^\sigma = Q \times S$,*

$$P^\sigma((q,s),(q',s')) = \begin{cases} \sum_{\alpha \in Act(s)} act(q,s,\alpha) \cdot P(s,\alpha,s') & \text{if } q' = mode(q,s) \\ 0 & \text{if } q' \neq mode(q,s) \ , \end{cases}$$

with $L^\sigma(q,s) = L(s)$ and $rew^\sigma(q,s) = rew(s)$, for all $q \in Q$ and $s \in S$.

If σ is memoryless, we sometimes omit its mode and write (s) instead of (q,s). For the MDPR in Fig. 1b and a scheduler that chooses action α in states s_0, s_1 and action τ in states s_2, s_3, the induced DTMCR is shown in Fig. 1c.

Different executions in several models can be seen as executions in the composition of the models. To simplify notation, in this paper we restrict ourselves to comparing executions in the same model, leading to the notion of *self-composition*.

Definition 6. *Assume an MDPR $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$ and a sequence $\boldsymbol{\sigma} = (\sigma_0, \ldots, \sigma_{n-1}) \in (\Sigma^{\mathcal{M}})^n$ of schedulers for $\mathcal{M}$. For $i \in \{0, \ldots, n-1\}$, let $\mathcal{M}_i = (S, Act, P, \mathsf{AP}_i, L_i, rew)$ with $\mathsf{AP}_i = \{a_i \mid a \in \mathsf{AP}\}$, and $L_i : S \to 2^{\mathsf{AP}_i}$ with $L_i(s) = \{a_i \mid a \in L(s)\}$. We define the n-*ary self composition *of $\mathcal{M}$ under $\boldsymbol{\sigma}$ as the DTMCR $\mathcal{M}^{\boldsymbol{\sigma}} = (S^{\boldsymbol{\sigma}}, P^{\boldsymbol{\sigma}}, \mathsf{AP}^{\boldsymbol{\sigma}}, L^{\boldsymbol{\sigma}}, rew^{\boldsymbol{\sigma}}) = \mathcal{M}_0^{\sigma_0} \times \ldots \times \mathcal{M}_{n-1}^{\sigma_{n-1}}$.*

In the above definition, $\mathcal{M}_i^{\sigma_i}$ is the DTMCR induced by $\mathcal{M}_i$ and σ_i. Note that the reward of a state $\boldsymbol{s} = ((q_0, s_0), \ldots, (q_{n-1}, s_{n-1})) \in S^{\boldsymbol{\sigma}}$ in the n-ary self-composition $\mathcal{M}^{\boldsymbol{\sigma}}$ is the sequence $rew^{\boldsymbol{\sigma}}(\boldsymbol{s}) = (rew(s_0), \ldots, rew(s_{n-1}))$, i.e. the ith state reward in the jth execution is $rew^{\boldsymbol{\sigma}}_{j,i}(\boldsymbol{s}) = rew_i(s_j)$. For a finite path π in $\mathcal{M}^{\boldsymbol{\sigma}}$, we denote its cumulative ith reward in the jth execution as $rew_{j,i}(\pi) = \sum_{k=0}^{|\pi|-1} rew_{j,i}(\pi[k])$.

3 HyperPCTL with Rewards

3.1 HyperPCTL Syntax

$$
\begin{aligned}
\varphi^{q} &::= \forall\hat{\sigma}.\varphi^{q} \mid \exists\hat{\sigma}.\varphi^{q} \mid \varphi^{sq} \\
\varphi^{sq} &::= \forall\hat{s}(\hat{\sigma}).\varphi^{sq} \mid \exists\hat{s}(\hat{\sigma}).\varphi^{sq} \mid \varphi^{nq} \\
\varphi^{nq} &::= \texttt{true} \mid a_{\hat{s}} \mid \varphi^{nq} \wedge \varphi^{nq} \mid \neg\varphi^{nq} \mid \varphi^{ar} \sim \varphi^{ar} \\
\varphi^{ar} &::= \mathbb{P}(\varphi^{path}) \mid \mathcal{R}_{\hat{s},i}(\varphi^{path}) \mid f(\varphi^{ar},\ldots,\varphi^{ar}) \\
\varphi^{path} &::= \bigcirc\varphi^{nq} \mid \varphi^{nq}\,\mathcal{U}\,\varphi^{nq} \mid \varphi^{nq}\,\mathcal{U}^{[k_1,k_2]}\,\varphi^{nq}
\end{aligned}
$$

Fig. 2: HyperPCTL syntax

Hyperproperties of executions in an MDPR can be specified using the logic HyperPCTL. As shown in Fig. 2, a *quantified formula* φ^{q} starts with a sequence of quantifiers over scheduler variables $\hat{\sigma} \in \hat{\Sigma}$, fixing the schedulers under which executions are considered. Inside, a *state-quantified formula* φ^{sq} defines a sequence of quantifiers over state variables $\hat{s} \in \hat{S}$, where each quantifier specifies a new execution from a given state under a given scheduler. Note that different executions might use the same scheduler.

In the scope of these quantifiers is a *non-quantified state formula* φ^{nq}, which can be the constant `true`, an atomic proposition indexed with a state variable, a conjunction, a negation, or a relational constraint comparing two arithmetic expressions via $\sim \in \{>, \geq, =, \neq, <, \leq\}$. *Arithmetic expressions* are constructed from probability expressions, reward expressions or applying arithmetic function symbols (e.g., addition, subtraction, multiplication, etc., where constants are 0-ary functions) to arithmetic expressions. Note that the reward operator $\mathcal{R}$ is indexed with a state variable $\hat{s}$ specifying the execution for which we consider the reward, and an integer i specifying the reward component; for models with unary rewards, like in our examples, we skip the second index (as it is always 0). Finally, the parameters of probabilistic and reward expressions are *path formulas*, which apply one of the temporal operators, next ($\bigcirc$), unbounded until ($\mathcal{U}$), or bounded until ($\mathcal{U}^{[k_1,k_2]}$, $k_1 \leq k_2 \in \mathbb{N}_{\geq 0}$) to non-quantified state formulas.

A HyperPCTL *formula* is a quantified formula in that every occurrence of an indexed atomic proposition $a_{\hat{s}}$ is in the scope of a state quantifier for $\hat{s}(\hat{\sigma})$, which in turn is in the scope of a scheduler quantifier for $\hat{\sigma}$. W.l.o.g., in the following we assume that each scheduler or state variable is quantified at most once.

In addition to standard syntactic sugar $\vee, \rightarrow, \Diamond, \Box, \ldots$, we can express expected cumulative reward over the next $t \in \mathbb{N}$ steps and expected reward in the state reached after t steps as follows:

$$
\mathcal{R}_{\hat{s},i}(C_t) = \mathcal{R}_{\hat{s},i}(\texttt{true}\,\mathcal{U}^{[t,t]}\texttt{true}) \text{ and } \mathcal{R}_{\hat{s},i}(I_t) = \begin{cases} \mathcal{R}_{\hat{s},i}(C_t) - \mathcal{R}_{\hat{s},i}(C_{t-1}) & \text{if } t{>}0 \\ \mathcal{R}_{\hat{s},i}(C_t) & \text{else .} \end{cases}
$$

3.2 HyperPCTL Semantics

HyperPCTL formulas are evaluated recursively in the context of an MDPR $\mathcal{M}$, a sequence $\boldsymbol{\sigma}$ of actions and a sequence $\boldsymbol{s}$ of states, both of the same length. Intuitively, the length of these sequences says how many executions we run in parallel, and the ith elements in these sequences specify the ith execution of the scheduler and the initial state in the induced DTMCR, respectively. An MDPR $\mathcal{M}$ satisfies a HyperPCTL formula φ (written $\mathcal{M} \models \varphi$) iff $\mathcal{M}, (), () \models \varphi$.

In the semantic rules below, the substitution $\varphi[\hat{\sigma} \rightsquigarrow \sigma]$ remembers the instantiation of a scheduler variable $\hat{\sigma}$ by a scheduler $\sigma = (Q, act, mode, init)$ through

syntactically transforming in φ each $\forall\hat{s}(\hat{\sigma})$ and $\exists\hat{s}(\hat{\sigma})$ into $\forall\hat{s}(\sigma)$ and $\exists\hat{s}(\sigma)$, resp. When instantiating the nth state quantifier $\forall\hat{s}(\sigma)$ or $\exists\hat{s}(\sigma)$ by a state s, we "start" an nth execution in state $(init(s), s)$ of $\mathcal{M}^\sigma$, which corresponds to extending the previously $(n-1)$-ary self-composition of $\mathcal{M}$ to *arity* n. We remember this by adding σ and s at the end of the corresponding sequences in the context (using concatenation $\circ$), and applying the substitution $\varphi[\hat{s}{\rightsquigarrow}n]$ to replace each indexed atomic proposition $a_{\hat{s}}$ and each reward operator $\mathcal{R}_{\hat{s},i}$ in φ by a_n and $\mathcal{R}_{n,i}$, respectively.[4] We recall from [2] the semantics of constructs that are not related to rewards:

$$
\begin{array}{ll}
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \forall\hat{\sigma}.\varphi & \textit{iff } \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi[\hat{\sigma}{\rightsquigarrow}\sigma] \textit{ for all } \sigma \in \Sigma^{\mathcal{M}} \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \exists\hat{\sigma}.\varphi & \textit{iff } \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi[\hat{\sigma}{\rightsquigarrow}\sigma] \textit{ for some } \sigma \in \Sigma^{\mathcal{M}} \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \forall\hat{s}(\sigma).\varphi & \textit{iff } \mathcal{M},\boldsymbol{\sigma}\circ\sigma,\boldsymbol{s}\circ(init(s),s) \models \varphi[\hat{s}{\rightsquigarrow}|\boldsymbol{\sigma}|] \textit{ for all } s{\in}S \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \exists\hat{s}(\sigma).\varphi & \textit{iff } \mathcal{M},\boldsymbol{\sigma}\circ\sigma,\boldsymbol{s}\circ(init(s),s) \models \varphi[\hat{s}{\rightsquigarrow}|\boldsymbol{\sigma}|] \textit{ for some } s{\in}S \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \texttt{true} & \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models a_i & \textit{iff } a_i \in L^{\boldsymbol{\sigma}}(\boldsymbol{s}) \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi_1 \wedge \varphi_2 & \textit{iff } \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi_1 \textit{ and } \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi_2 \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \neg\varphi & \textit{iff } \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \not\models \varphi \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi_1^{ar} \sim \varphi_2^{ar} & \textit{iff } [\![\varphi_1^{ar}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} \sim [\![\varphi_2^{ar}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} \\
[\![\mathbb{P}(\varphi_{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = \Pr^{\mathcal{M}^{\boldsymbol{\sigma}}}\big(\{\pi \in Paths_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma}}} \mid \mathcal{M},\boldsymbol{\sigma},\pi \models \varphi_{path}\}\big) \\
[\![f(\varphi_1^{ar},\ldots,\varphi_k^{ar})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = f\big([\![\varphi_1^{ar}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}},\ldots,\ [\![\varphi_k^{ar}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}}\big)
\end{array}
$$

We are left with the semantics for $\mathcal{R}_{j,i}(\varphi^{path})$ (note that instantiating a state quantifier for $\hat{s}(\sigma)$ replaces each $\mathcal{R}_{\hat{s},i}$ occurrence by $\mathcal{R}_{j,i}$, where j is the position of the quantifier). The value of $\mathcal{R}_{j,i}(\bigcirc\varphi^{nq})$ is the current ith reward plus the expected ith reward of the successor state in the jth execution, if the probability that the successor state satisfies φ^{nq} is 1; otherwise, the value is undefined. The value of $\mathcal{R}_{j,i}(\varphi_1^{nq}\ \mathcal{U}\ \varphi_2^{nq})$ is the expected cumulative ith reward in the jth execution, accumulated until the first time a (global self-composition) state is reached that satisfies φ_2^{nq}, in case the probability of satisfying $\varphi_1^{nq}\ \mathcal{U}\ \varphi_2^{nq}$ is 1; otherwise, the value is undefined. The semantics of $\mathcal{R}_{j,i}(\varphi_1^{nq}\ \mathcal{U}^{[k_1,k_2]}\ \varphi_2^{nq})$ is similar, but the rewards are accumulated until the first satisfaction of φ_2^{nq} within time $[k_1,k_2]$. Formally, the semantics for $[\![\mathcal{R}_{j,i}(\varphi^{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}}$ is as follows, given that $[\![\mathbb{P}(\varphi^{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} = 1$. If that is not the case, $[\![\mathcal{R}_{j,i}(\varphi^{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}}$ is undefined.

$$
\begin{array}{ll}
fPaths_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma}}}(\varphi_1^{nq}\,\mathcal{U}\,\varphi_2^{nq}) & = \{\boldsymbol{s}_0\ldots\boldsymbol{s}_n \in fPaths_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma}}} \mid \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}_n \models \varphi_2^{nq} \textit{ and} \\
 & \quad \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}_i \models \varphi_1^{nq} \wedge \neg\varphi_2^{nq} \textit{ for } i = 0,\ldots,n{-}1\} \\
fPaths_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma}}}(\varphi_1^{nq}\,\mathcal{U}^{[k_1,k_2]}\varphi_2^{nq}) & = \{\boldsymbol{s}_0\ldots\boldsymbol{s}_n \in fPaths_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma}}} \mid k_1 \le n \le k_2 \textit{ and} \\
 & \quad \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}_n \models \varphi_2^{nq} \textit{ and} \\
 & \quad \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}_i \models \varphi_1^{nq} \textit{ for } i = 0,\ldots,k_1{-}1 \textit{ and} \\
 & \quad \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}_i \models \varphi_1^{nq} \wedge \neg\varphi_2^{nq} \textit{ for } i = k_1,\ldots,n{-}1\} \\
[\![\mathcal{R}_{j,i}(\bigcirc\varphi^{nq})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = rew_{j,i}^{\boldsymbol{\sigma}}(\boldsymbol{s}) + \textstyle\sum_{\boldsymbol{s}'\in S^{\boldsymbol{\sigma}}} P^{\boldsymbol{\sigma}}(\boldsymbol{s},\boldsymbol{s}')\cdot rew_{j,i}^{\boldsymbol{\sigma}}(\boldsymbol{s}') \\
[\![\mathcal{R}_{j,i}(\varphi_1^{nq}\,\mathcal{U}\,\varphi_2^{nq})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = \textstyle\sum_{\pi\in fPaths_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma}}}(\varphi_1^{nq}\,\mathcal{U}\,\varphi_2^{nq})}(\Pr^{\boldsymbol{\sigma}}(\pi)\cdot rew_{j,i}^{\boldsymbol{\sigma}}(\pi)) \\
[\![\mathcal{R}_{j,i}(\varphi_1^{nq}\,\mathcal{U}^{[k_1,k_2]}\varphi_2^{nq})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = \textstyle\sum_{\pi\in fPaths_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma}}}(\varphi_1^{nq}\,\mathcal{U}^{[k_1,k_2]}\varphi_2^{nq})}(\Pr^{\boldsymbol{\sigma}}(\pi)\cdot rew_{j,i}^{\boldsymbol{\sigma}}(\pi))
\end{array}
$$

[4] Instead of syntactical substitutions, we could also use binding functions to map scheduler variables to schedulers and state variables to indices in the state sequence in the context.

Since adding rewards to HyperPCTL causes arithmetic values to be potentially undefined, we need to extend the above semantics to handle the propagation of undefined values. For each syntactic case, the above semantics remains unchanged if all involved statements used in the definition are defined. It would be an easy job to set the values in all other cases to undefined. However, even if some of the arguments are undefined, we still might be able to conclude a defined value. For example, if one of the operands in a conjunction is false then the conjunction is inevitably false, even if the other operand is undefined. In extension to the above semantics for the cases when all terms used in the definition are defined, below we fix the semantics for the remaining cases with the objective to reduce the occurrence of undefined values.

We extend the Boolean domain of true (1) and false (0) with undefined ($\bot$). We use the $\models$ relation as before when all sub-expressions (and thus the formula) are known to be defined, and use $[\![\cdot]\!]$. otherwise. Logical constants as well as atomic propositions are always defined. The value of a conjunction is undefined iff none of the operands is false and not both operands are true, whereas a negation is undefined iff the negated formula is undefined.

The value of a universally state-quantified formula $\forall \hat{s}(\sigma).\varphi$ is undefined if the value of φ is undefined for at least one instantiation of the formula with a state and is not false for any other instantiation. Likewise, the value of an existentially state-quantified formula $\exists \hat{s}(\sigma).\varphi$ is undefined if the value of φ is undefined for at least one instantiation of the formula with a state and is not true for any other instantiation. The undefinedness of scheduler quantifiers is analogous.

Also the domain of arithmetic values gets extended with the undefined value $\bot$. Arithmetic function applications $f(\varphi_1, \ldots, \varphi_k)$ and arithmetic constraints $\varphi_1 \sim \varphi_2$ are undefined iff any of their parameters are undefined. However, for probabilistic until $\varphi = \mathbb{P}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ we can exploit available information to increase the number of defined cases, even if the satisfaction of one of the operands is undefined in the current state, as shown in Table 1. The information we exploit for the semantics in a state s are the probabilistic until values in the successor states, or more precisely, the value of $p = \sum_{s' \in S^\sigma} P(s, s') \cdot [\![\varphi]\!]_{\mathcal{M},\sigma,s'} \in [0,1] \cup \{\bot\}$, which we consider undefined iff one of the successor probabilities is undefined.

Row	$[\![\varphi_1]\!]$	$[\![\varphi_2]\!]$	p	$[\![\varphi]\!]$
1	*	1	*	1
2	0	0	*	0
3	$\bot$	0	0	0
4	$\bot$	0	$\neq 0$	$\bot$
5	1	0	*	p
6	0	$\bot$	*	$\bot$
7	$\bot$	$\bot$	*	$\bot$
8	1	$\bot$	1	1
9	1	$\bot$	$\neq 1$	$\bot$

Table 1: Semantics of $\varphi = \mathbb{P}(\varphi_1 \,\mathcal{U}\, \varphi_2)$, partly depending on $p = \sum_{s' \in S^\sigma} P(s, s') \cdot [\![\varphi]\!]_{\mathcal{M},\sigma,s'} \in [0,1] \cup \{\bot\}$. Here, $[\![.]\!]$ is short for $[\![.]\!]_{\mathcal{M},\sigma,s}$.

Table 1 extends the original probabilistic until semantics from above with the undefined cases, using $*$ to denote an arbitrary (defined or undefined) arithmetic value. This table is split into three parts. The first part states that if φ_2 is true then the formula value is 1. The second part covers the case where φ_2 is false, where the violation of φ_1 leads to the violation of the formula, and if φ_1 is true then the formula probability equals the value of p. An interesting case

in the second block is when φ_1 is undefined: though in most cases the formula is also undefined, if we know that the probability to satisfy the until formula in the future is 0 then we can safely state that the probability to satisfy the same in the current state is also 0. Similarly in the third block, if φ_1 is true in the current state and the probability to satisfy the until formula in the future is 1 then, irrelevant of the value of φ_2, the probability to satisfy the until formula from the current state is always 1.

Reward expressions are undefined if the respective path property is not satisfied with probability 1. For the reward expression $\mathcal{R}_{j,i}(\bigcirc\varphi)$, this is the only case in which it is undefined. To evaluate $\varphi = \mathcal{R}_{j,i}(\varphi_1 \,\mathcal{U}\, \varphi_2)$, if φ_2 is true in the current state then we need to know only the current state's reward; in this case the reward is defined independent of the successor states. If φ_2 is false currently then the reward is computed from the current state reward plus the expected successor φ-values, thus undefinedness of the reward expression in a successor state causes undefinedness in the current state. However, if φ_2 is undefined in the current state then we do not know which of these two cases apply; the only case where this does not matter is if the reward expression evaluates in all successor states to 0, namely then the value of φ is the current state reward. Thus if φ_2 is undefined in the current state then the reward expression is undefined in all but this special case, even if the probability of the until formula is 1.

The definedness of bounded until formulas is similar to the unbounded case for both probability and reward expressions, except that we now also need to account for the bounds.

However, with these definitions, we only exploit some but not all information, to determine the definedness of a property. Assume, for example, the property that from a state s, the probability to eventually satisfy φ is less than p. It might be the case that in some states reachable from s the value of φ is undefined, triggering the above probability to be undefined by our algorithm. However, φ might be reachable along another path with a probability larger than p, in which case we could have safely stated that it is at least p. Hence, it can be a direction of future research to find a tighter bound on the definedness of a property.

4 Applications of HyperPCTL with Rewards

4.1 Timing Attacks

Side-channel timing leaks can potentially reveal sensitive information. For example, RSA uses the modular exponentiation algorithm on the right to compute $a^b \ mod\ n$, where a is the message and b is the encryption key. This implementation is flawed because of the *if* in line 6. Due to the lack of an *else* branch, its execution will take longer if b contains more 1-bits. An attacker could therefore run a thread in parallel to measure the execution time of the algorithm to derive the

```
1  void mexp(){
2    c = 0; d = 1; i = k;
3    while (i >= 0){
4      i = i-1; c = c*2;
5      d = (d*d) % n;
6      if (b(i) = 1){
7        c = c+1;
8        d = (d*a) % n;
9      }
10   }
11 }
```

number of 1-bits in the encryption key. To prevent such vulnerabilities, we would like the execution time to be independent of the bit values in the encryption key, which is captured by assigning a reward of 1 to each state in the MDPR. Here, each state represents the current position in the code and loop iteration. This results in the following HyperPCTL formula:

$$\forall\hat{\sigma}_1.\forall\hat{\sigma}_2.\forall\hat{s}(\hat{\sigma}_1).\forall\hat{s}'(\hat{\sigma}_2).\,(init_{\hat{s}} \wedge init_{\hat{s}'}) \rightarrow (\mathcal{R}_{\hat{s}}(\Diamond\, end_{\hat{s}}) = \mathcal{R}_{\hat{s}'}(\Diamond\, end_{\hat{s}'})).$$

4.2 Probabilistic Conformance

The aim here is to ensure that an implementation conforms with the system it is simulating [2]. We consider the implementation of a 6-sided die with repeated tossing of a fair coin using the Knuth-Yao algorithm [22]. For conformance, the probabilistic distribution of reaching the 6 sides of a die should be equal in both cases. We model this problem with an MDP consisting of two components: the first component describes the die and its states represent the faces of the die after being rolled. The second component describes the multiple coin tosses and its states represent the unique combined results of the tosses. Extending this model with rewards allows us to synthesize *efficient* implementations: if we assign to every state, except the absorbing states, a reward of 1, the expected reward on reaching one of the absorbing states in the coin implementation will be equal to the expected number of coin tosses in it. If we limit the rewards collected in such a path, we can filter the implementations with minimum intermediate states. The following formula specifies that the expected number of coin tosses in such an implementation must be less than 4:

$$\exists\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\exists\hat{s}'(\hat{\sigma}).dieInit_{\hat{s}} \rightarrow \left(\varphi \wedge \mathcal{R}_{\hat{s}'}(\Diamond(\bigvee_{l=1}^{6}(die = l)_{\hat{s}'})) < 4\right)$$

with $\varphi = coinInit_{\hat{s}'} \wedge \bigwedge_{l=1}^{6}(\mathbb{P}(\Diamond(die = l)_{\hat{s}}) = \mathbb{P}(\Diamond(die = l)_{\hat{s}'}))$.

4.3 Cost Analysis in Multi-Agent Path Planning

We consider the examples in Fig. 3 where two robots R_1, R_2 aim to reach the target cell *end* starting their journey from two different initial cells ($start_1$, $start_2$). The robots' behavior is modeled as an MDPR where each cell occupied represents a state. Nondeterministic actions represent all possible moves of the robot from each cell, while the successful maneuvering after having executed an action is captured by a probability distribution.

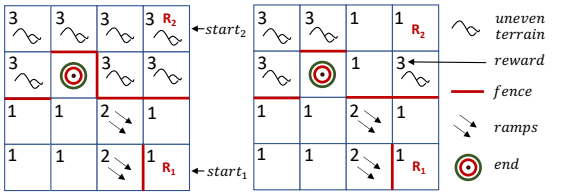


Fig. 3: The maze on the left satisfies φ_{target}, while on the right it violates φ_{target}.

Fences prevent a robot to move in a certain direction disabling possible actions in a particular cell, while the presence of ramps or uneven terrain can

increase/decrease the probability of correct robot maneuvers. The occupancy of each state has a cost in terms of energy consumption modeled as a positive reward. We want to check that for all possible (memoryless) schedulers, when robots R_1, R_2 start their mission from their respective initial conditions and they can both reach the target state with probability 1, then the expected energy consumption for robot R_1 is less than the expected energy consumption for robot R_2. This can be expressed as the following probabilistic hyperproperty:

$$\varphi_{target} = \forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\psi \rightarrow \Big(\mathcal{R}_{\hat{s}}(\diamondsuit\, end_{\hat{s}}) < \mathcal{R}_{\hat{s}'}(\diamondsuit\, end_{\hat{s}'})\Big), \text{ where}$$

$$\psi = \Big(start_{1\,\hat{s}} \wedge start_{2\,\hat{s}'} \wedge \mathbb{P}(\diamondsuit\, end_{\hat{s}}) = 1 \wedge \mathbb{P}(\diamondsuit\, end_{\hat{s}'}) = 1\Big).$$

4.4 Probabilistic Self-stabilizing Systems

In distributed systems, randomization is often used to break symmetry between processes to tackle impossibility results. For instance, self-stabilizing token circulation in a ring is impossible in a non-probabilistic setting but Herman's algorithm [20] (see Fig. 4) uses randomization to ensure recovery to a *stable state* (i.e., there is only one token circulating) with probability one. In such an algorithm, from certain initial states, convergence to a stable state may be faster than others and if faults hit those states with a higher probability, it reduces the average convergence time significantly. Thus, designers of self-stabilizing algorithms often use state encodings to tackle slow recovery [11]. The following formula intends to check whether there exists a state from which the convergence time is twice slower than from some other state:

$$\forall\hat{\sigma}.\exists\hat{s}(\hat{\sigma}).\exists\hat{s}'(\hat{\sigma}).\Big(\mathcal{R}_{\hat{s}}(\diamondsuit\, stable_{\hat{s}}) > 2 \cdot \mathcal{R}_{\hat{s}'}(\diamondsuit\, stable_{\hat{s}'})\Big)$$

Note that Herman's algorithm yields a DTMCR and, thus, the choice of scheduler quantification is irrelevant.

1: **Variable:** x_i : *boolean* $\in \{0, 1\}$
2: **Guarded Commands:**
$x_i = x_{i-1} \longrightarrow p : x_i := 0 \; + \; (1-p) : x_i := 1;$
$x_i \neq x_{i-1} \longrightarrow 1 : x_i := x_{i-1};$

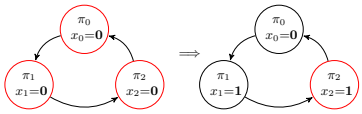


Fig. 4: Herman's algorithm [20] for process i and example for three processes.

5 Model Checking Algorithm for Reward Operators

HyperPCTL provides an increased level of expressiveness over PCTL and PCTL*, causing the model checking problem for MDPRs to be undecidable even without rewards, as shown in [2]. To achieve decidability for HyperPCTL without rewards, in [2] we restricted the domain of scheduler quantification to *memoryless non-probabilistic schedulers*. For this restricted domain, the model checking problem is NP-complete (or coNP-complete) when the scheduler quantification is existential (or universal). We provided a model checking algorithm by logically encoding

HyperPCTL satisfaction problems as linear real-arithmetic formulas and use an SMT solver to check the encodings for satisfiability. Elaborate explanations of encoding non-reward operators can be found in [2].

After adding rewards, the model checking problem restricted to finite memoryless schedulers is still decidable. Similar to the standard model checking problem for Markov Reward Models, computing the expected reward earned until a certain set of states is reached, has a polynomial time complexity in the size of the MDP: the problem can be solved by determining a linear real-arithmetic equation system via graph reachability analysis and solving it. This means adding rewards does not change the class of complexity of the model checking problem as identified in [2].

However, adding rewards to the problem requires a major adaption of the logical encoding. The reason is that expected reward values might be undefined, and undefinedness might propagate from the inner sub-formulas to the formula value. The main contributions of this section are (1) to extend the model checking algorithm from [2] to encode the semantics of reward-related HyperPCTL expressions and (2) to modify the previous encodings to model undefinedness propagation for the remaining language components. To ease understanding, in the following we consider unary-reward models and a single existential scheduler quantifier in our properties; extension to multi-dimensional rewards and several scheduler quantifiers without quantifier alternation is doable by little modifications to the algorithms. Given their finite domain, support for scheduler quantifier alternation is possible, too, but it would require more involved extensions.

Algorithm 1: Main SMT encoding algorithm

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR;
φ: HyperPCTL formula.
Output: Whether $\mathcal{M}$ satisfies φ.

1 **Function** $\texttt{Main}(\mathcal{M}, \varphi = \exists\hat{\sigma}.Q_1\hat{s}_1(\hat{\sigma}).\ldots Q_n\hat{s}_n(\hat{\sigma}).\varphi^{nq})$**:**
2 $\quad E := \bigwedge_{s\in S}(\bigvee_{\alpha\in Act(s)} \sigma_s = \alpha)$
3 $\quad E := E \wedge \texttt{Semantics}(\mathcal{M}, \varphi^{nq}, n)$
4 $\quad T := E \wedge \texttt{Eval}(\mathcal{M},\ \varphi,\ \{1\})$
5 $\quad U := E \wedge \texttt{Eval}(\mathcal{M},\ \varphi,\ \{\bot, 1\})$
6 $\quad$ **if** $check(T) = SAT$ **then return** $TRUE$
7 $\quad$ **else if** $check(U) = SAT$ **then return** $UNDEF$
8 $\quad$ **else return** $FALSE$

Assume as input an MDPR model $\mathcal{M}$ and a HyperPCTL formula φ. In [2] we used Boolean variables $holds_{s,\varphi}$ to encode the truth value of a Boolean-valued formula φ in state s. In this work, we replace the two-valued domain for these variables by a three-valued domain over the values *true* (1), *false* (0) and *undefined* ($\bot$). Furthermore, we use variables $val_{s,\varphi}$ to store the numerical value of an arithmetic expression φ in state s. To also encode the definedness of arithmetic values, we introduce additional Boolean variables $def_{s,\varphi}$ which should be true iff the corresponding value is defined. Finally, to encode a scheduler, we use for each state of $\mathcal{M}$ a variable σ_s to store the chosen action.

Algorithm 2: SMT encoding for the meaning of an input formula

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR; φ: quantifier-free HyperPCTL formula or expression; n: number of state variables in φ.
Output: SMT encoding of the meaning of φ in n-ary self-composition of $\mathcal{M}$.

1 **Function** `Semantics`$(\mathcal{M}, \varphi, n)$**:**
2 **if** φ *is true* **then** $E := \bigwedge_{s \in S^n} holds_{s,\varphi}{=}1$
3 **else if** φ *is* $a_{\hat{s}_i}$ **then**
4 $E := (\bigwedge_{s \in S^n,\, a \in L(s_i)} (holds_{s,\varphi}{=}1)) \wedge (\bigwedge_{s \in S^n,\, a \notin L(s_i)} (holds_{s,\varphi}{=}0))$
5 **else if** φ *is* $\neg\varphi'$ **then**
6 $E :=$ `Semantics`$(\mathcal{M}, \varphi', n) \wedge \bigwedge_{s \in S^n} (holds_{s,\varphi'}{=}0 \rightarrow holds_{s,\varphi}{=}1) \wedge$
7 $\bigwedge_{s \in S^n} (holds_{s,\varphi'}{=}1 \rightarrow holds_{s,\varphi}{=}0) \wedge \bigwedge_{s \in S^n} (holds_{s,\varphi'}{=}\bot \rightarrow holds_{s,\varphi}{=}\bot)$
8 **else if** φ *is* $\varphi_1 \wedge \varphi_2$ **then** $E :=$ `SemanticsConjunction`$(\mathcal{M}, \varphi, n)$
9 **else if** φ *is* $\varphi_1^{ar} \sim \varphi_2^{ar}$ **then** $E :=$ `SemanticsComp`$(\mathcal{M}, \varphi, n)$
10 **else if** φ *is* $f(\varphi_1^{ar}, \ldots, \varphi_k^{ar})$ **then** $E :=$ `SemanticsArithmetic`$(\mathcal{M}, \varphi, n)$
11 **else if** φ *is* $\mathbb{P}(\bigcirc \varphi')$ **then** $E :=$ `SemanticsNext`$(\mathcal{M}, \varphi, n)$
12 **else if** φ *is* $\mathbb{P}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ **then** $E :=$ `SemanticsUnboundedUntil`$(\mathcal{M}, \varphi, n)$
13 **else if** φ *is* $\mathbb{P}(\varphi_1 \,\mathcal{U}^{[k_1,k_2]}\, \varphi_2)$ **then** $E :=$ `SemanticsBoundedUntil`$(\mathcal{M}, \varphi, n)$
14 **else** $E :=$ `RewSemantics`$(\mathcal{M}, \varphi, n)$
15 **return** E

The starting point of the encoding is Algorithm 1, which begins by encoding the scheduler choice [5] in line 2. The semantics of the non-quantified inner formula φ^{nq} under a given scheduler choice in each of the states is encoded in line 3. This basic encoding E is extended in two directions: formula T encodes that φ can be made true by some suitable quantifier instantiation, whereas U encodes that φ can be made true or undefined. Only if none of these two cases apply (i.e. if both formulas are unsatisfiable), we conclude that $\mathcal{M}$ does not satisfy φ. Not listed in the algorithm is the case of a universal scheduler quantifier, where we use negation to get an existential formula, apply the listed algorithm, and negate the answer.

The semantics of formulas is encoded by Algorithm 2. We omit the pseudocode of sub-algorithms that were needed also without rewards; these are similar to those in [2] but get extended with the encoding of definedness as explained in Section 3.2. Relevant for rewards is line 14, calling the method `RewSemantics` in Algorithm 3 to encode the semantics of the reward operators. In the case of rewards over the next operator $\varphi = \mathcal{R}_{\hat{s}_i}(\bigcirc \varphi')$, we first encode the probability $\mathbb{P}(\bigcirc \varphi')$; φ is undefined if this probability is not 1 (line 5). If the probability is defined, then the reward is the expected reward of the successors in the ith execution (line 7).

[5] For n scheduler quantifiers, we would simply need to include such a scheduler encoding for each of the schedulers $\sigma_1, \ldots, \sigma_n$, and in the rest of the encoding, refer to the respective schedulers σ_i instead of σ.

Algorithm 3: SMT encoding for the meaning of reward operators

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR; φ: quantifier-free HyperPCTL formula or expression; n: number of state variables in φ.
Output: SMT encoding of the meaning of φ in n-ary self-composition of $\mathcal{M}$.

Function `RewSemantics`$(\mathcal{M}, \varphi, n)$**:**
  if φ *is* $\mathcal{R}_{\hat{s}_i}(\bigcirc\varphi')$ **then**
    $E :=$ `Semantics`$(\mathcal{M}, \mathbb{P}(\bigcirc\varphi'), n)$
    foreach $s = (s_1, \ldots, s_n) \in S^n$ **do**
      $E := E \wedge ((val_{s,\mathbb{P}(\bigcirc\varphi')} \neq 1 \vee \neg def_{s,\mathbb{P}(\bigcirc\varphi')}) \leftrightarrow \neg def_{s,\varphi})$
      foreach $\alpha = (\alpha_1, \ldots, \alpha_n) \in Act(s_1) \times \ldots \times Act(s_n)$ **do**
        $E := E \wedge ([def_{s,\varphi} \wedge \bigwedge_{j=1}^{n} \sigma_{s_j} = \alpha_j] \rightarrow [val_{s,\varphi} = rew(s_i) + \sum_{s' \in supp(\alpha_1)\times\ldots\times supp(\alpha_n)}((\prod_{j=1}^{n} P(s_j, \alpha_j, s'_j)) \cdot rew(s'_i))])$
  else if φ *is* $\mathcal{R}_{\hat{s}_i}(\varphi_1 \,\mathcal{U}^{\,[k_1,k_2]}\, \varphi_2)$ **then**
    $E :=$ `SemanticsBoundedUntil`$(\mathcal{M}, \mathbb{P}(\varphi_1 \,\mathcal{U}^{\,[k_1,k_2]}\, \varphi_2), n)$
    $E := E \wedge$ `RewardBoundedUntil`$(\mathcal{M}, \varphi, n)$
  else if φ *is* $\mathcal{R}_{\hat{s}_i}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ **then**
    $E :=$ `SemanticsUnboundedUntil`$(\mathcal{M}, \mathbb{P}(\varphi_1 \,\mathcal{U}\, \varphi_2), n)$
    $E := E \wedge$ `RewardUnboundedUntil`$(\mathcal{M}, \varphi, n)$
  return E

Algorithm 4: SMT encoding for reward of unbounded until

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR; φ: HyperPCTL unbounded until formula of the form $\mathcal{R}_{\hat{s}_i}(\varphi_1 \mathcal{U} \varphi_2)$; n: number of state variables in φ.
Output: SMT encoding of φ's meaning in the n-ary self-composition of $\mathcal{M}$.

Function `RewardUnboundedUntil`$(\mathcal{M}, \varphi = \mathcal{R}_{\hat{s}_i}(\varphi_1 \mathcal{U} \varphi_2), n)$**:**
  $\varphi' := \mathbb{P}(\varphi_1 \,\mathcal{U}\, \varphi_2)$; $E :=$ `true`
  foreach $s = (s_1, \ldots, s_n) \in S^n$ **do**
    $E := E \wedge (holds_{s,\varphi_2} = 1 \rightarrow (val_{s,\varphi} = rew(s_i) \wedge def_{s,\varphi}))$
    $E := E \wedge ((val_{s,\varphi'} \neq 1 \vee \neg def_{s,\varphi'}) \rightarrow \neg def_{s,\varphi})$
    foreach $\alpha = (\alpha_1, \ldots, \alpha_n) \in Act(s_1) \times \ldots \times Act(s_n)$ **do**
      $E := E \wedge ((val_{s,\varphi'} = 1 \wedge def_{s,\varphi'} \wedge holds_{s,\varphi_2} \neq 1 \wedge \bigwedge_{j=1}^{n} \sigma_{s_j} = \alpha_j) \rightarrow$
      $[val_{s,\varphi} =$
       $rew(s_i) + \sum_{s' \in supp(\alpha_1)\times\ldots\times supp(\alpha_n)}((\prod_{i=1}^{n} P(s_j, \alpha_j, s'_j)) \cdot val_{s',\varphi}) \wedge$
      $(\neg def_{s,\varphi} \leftrightarrow [(\bigvee_{s' \in supp(\alpha_1)\times\ldots\times supp(\alpha_n)} \neg def_{s',\varphi}) \vee$
      $(holds_{s,\varphi_2} = \bot \wedge val_{s,\varphi} \neq rew(s_i))])])$
  return E

To encode the reward of unbounded until formulas, we first need to encode the probability of the until formula, since this probability needs to be 1 for a defined reward value. Then we call the `RewardUnboundedUntil` method from Algorithm 4, which implements the semantics of the reward of unbounded until

Case study		VR	Running time (s)			#SMT variables	#sub formulas	#states	#transitions
			Encoding	Solving	Total				
TA	1-bit key	×	0.11	0.01	0.12	344	1008	8	10
	16-bit key	×	16.41	3.69	20.10	19244	49728	68	100
	30-bit key	×	143.49	44.64	188.13	62868	160160	124	184
	45-bit key	×	774.53	1304.98	2079.51	137448	348080	184	274
PC	s=(0)	✓	5.03	2.03	7.06	7281	34681	20	186
	s=(0,1,2)	✓	6.66	8.91	15.57	7281	61631	20	494
	s=(0,...,4)	✓	8.82	35	43.82	7281	88581	20	802
	s=(0,...,6)	✓	11.64	53.05	64.69	7281	115531	20	1110
RO	3x3	✓	0.87	0.05	0.92	2179	7622	18	66
	3x3	×	0.93	0.05	0.98	2179	7622	18	66
	4x4	✓	3.55	0.28	3.83	6561	21572	32	160
	4x4	×	3.43	0.25	3.68	6561	21476	32	148
	5x5	✓	13.07	0.5	13.57	15651	48302	50	250
	5x5	×	13.19	0.98	14.17	15651	48302	50	250
	6x6	✓	44.52	1.04	45.56	32041	96096	72	398
	6x6	×	44.65	7.48	52.13	32041	96096	72	398
HS	$n = 3$	✓	0.1	0.01	0.11	489	4655	8	28
	$n = 5$	✓	0.95	0.13	1.08	2369	7047	32	244
IJ	$n = 3$	✓	0.08	0.01	0.09	169	698	7	21
	$n = 4$	✓	0.24	0.04	0.28	601	2194	15	56
	$n = 5$	✓	0.89	0.33	1.22	2233	7010	31	140
	$n = 6$	✓	3.93	19.39	23.32	8569	23362	63	336

Table 2: Experimental results. **VR:** Verification result. **TA**: Timing attack. **PC**: Probabilistic conformance. **RO**: Robotics example. **HS**: Herman's algorithm. **IJ**: Israeli-Jaflon's algorithm. ✓: the result is true. ×: the result is false.

from Section 3.2. Undefinedness is covered in line 5, when the probability of the unbounded until is either not defined or not 1, and in the lines 9-10, when the probability of the unbounded until is 1, φ_2 is not true and either a successor reward is undefined, or φ_2 is undefined and the successor rewards are not zero. The method `RewardBoundedUntil` for reward expressions with bounded until, not shown here, is similar to the unbounded case, but needs additional bookkeeping about the time interval within which φ_2 needs to be satisfied.

6 Evaluation

We have implemented a prototype of the presented algorithm by extending our tool HyperProb [10] to support rewards. The implementation has been coded in Python using the libraries Lark [24] for parsing the input formula, and Stormpy [26] for parsing the input MDPR. The generated constraints are then solved by the SMT solver

Algorithm 5: Encoding certain formula values

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR; φ: HyperPCTL formula; $v \subseteq \{0, 1, \bot\}$.
Output: Encoding that $\mathcal{M}, (), () \models \exists \hat{\sigma}. Q_1 \hat{s}_1 \ldots Q_n \hat{s}_n. (\varphi^{nq} \in v)$.

1 **Function** `Eval`($\mathcal{M}$, $\varphi = \exists \hat{\sigma}. Q_1 \hat{s}_1 \ldots Q_n \hat{s}_n. \varphi^{nq}$, v):
2 **foreach** $i = 1, \ldots, n$ **do**
3 **if** $Q_i = \forall$ **then** $B_i :=$ "$\bigwedge_{s_i \in S}$" **else** $B_i :=$ "$\bigvee_{s_i \in S}$"
4 **return** $B_1 \ldots B_n$ $(holds_{(s_1,\ldots,s_n),\varphi^{nq}} \in v)$

Z3 [25]. Our implementation cannot handle all possible cases of undefinedness. We currently do not calculate the extent of partial definedness of a property in a model. We check whether the states queried in the property are reachable with a probability of one and proceed in calculation of rewards in such cases. Hence, we have evaluated case studies, where the reachability probabilities are always one.

The concept of rewards have eased the modeling of case studies with respect to counting of expected steps needed to reach a state. Hence, for timing attack and probabilistic conformance case studies, the number of transitions and states are less when compared to the models used in [2]. The implementation also returns a witness/counterexample whenever possible, allowing us to synthesize schedulers. Note that, though the ensemble of schedulers in the executions (i.e. $\boldsymbol{\sigma}$ in the semantical context) define a scheduler in the self-composition, not all schedulers of the self-composition can be defined this way, posing a major difference between scheduler synthesis for PCTL and for HyperPCTL.

For the **TA** case study, we have modeled the problem with {1, 16, 30, 45}-bit encryption keys. We have verified the HyperPCTL formula described in Section 4.1. The property does not hold on the given model and our implementation finds this bug. Since our implementation can handle only one scheduler quantifier, we have added a second copy of the model to the input MDPR such that the single scheduler can assign different actions to the states in the two copies of the model.

For the **PC** case study, we have verified the property described in Section 4.2. We have started with a model with all possible transitions, represented nondeterministically, from the initial state s_0. For all other states, we allowed only the transitions that will give us a correct solution. We challenged our implementation to synthesize a scheduler that will satisfy the required probabilities within the given reward bound. We scaled the model by incrementally allowing all possible combination of transitions using nondeterministic actions in each state and limited the expected coin tosses to be 4 for each experiment. For all the cases, our implementation was successful in finding a solution, which we verified manually as correct.

For the **RO** case study, we have verified the property described in Section 4.3. We have scaled the model in terms of maze size and verified both positive and negative cases of path finding. On self-stabilizing systems, we have verified several properties and described one of them in Section 4.4. This property is satisfied and we have successfully found a witness. We have reported the timing data for this property in Table 2. We have verified the property in models representing both Herman's (**HS**) and Israeli-Jaflon's (**IJ**) [21] algorithms. Since, Herman's algorithm is only valid for odd processes, we tried verification over $\{3, 5\}$ processes. For Israeli-Jaflon's, we tried it over $\{3, 4, 5, 6\}$ process.

The experiments have been performed in a Docker container running on a system with 2.3 GHz i7 processor and 32 GB of RAM. Because of the incomplete implementation of handling of undefined values, which would add a significant number of additional constraints, the reported execution times are lower than

they would normally be. From Table 2, it is clear that the execution times for even relatively small MPDRs are large. This is because of the inherent complexity of the problem, to which reward operators add a new dimension of complexity.

7 Related Work

The classical temporal logics for probabilistic systems [19], for example PCTL and its extension with reward operators [17, 23], cannot express probabilistic hyperproperties, because they can only refer to a single path at a time. There has been considerable work to overcome this shortcoming for non-probabilistic hyperlogics in terms of automated verification [8, 14–16] and monitoring [4, 6, 7, 12, 13, 18, 27] of HyperLTL specifications. However, none of these are relevant to probabilistic systems. The work in [3] overcomes this limitation by introducing HyperPCTL, a temporal logic that can express probabilistic hyperproperties over discrete-time Markov chains. In [1] we addressed the problem of computing the regions of parameter configurations of discrete-time Markov chains satisfying/violating a formula φ in a fragment of HyperPCTL. In [2], we enriched the syntax and semantics of HyperPCTL with the possibility to quantify simultaneously over schedulers and probabilistic computation trees. However, reasoning about rewards was not supported in [2], while it is considered in this paper for the first time.

An orthogonal attempt to solve the model checking problem has been addressed in [9], where the authors present the temporal logic PHL that allows quantification over schedulers, but path quantification of the induced DTMC is achieved by using HyperCTL*. To overcome the undecidability problem of model checking with their logics, the authors provide two approximate methods for proving and refuting only universally quantified formulas in PHL for memoryful schedulers. However, this work does not handle reward models as well.

Other works related to probabilistic hyperproperties comprises of approaches based on *statistical model checking* (SMC) [28, 29] using an extension of HyperPCTL that allows explicit path quantification over the probability operator. However, these approaches do not consider the use of rewards either.

8 Conclusion

In this paper, we studied probabilistic hyperproperties with rewards. To this end, we extended the temporal hyperlogic HyperPCTL with reward operators that associates quantified computation trees with interrelated accumulated rewards. We also proposed an SMT-based algorithm for model checking these formulas for MDPRs. We have created a prototypical implementation and used it to analyze a few case studies. Due to the high complexity of the problem, more efficient model checking algorithms are greatly needed. An orthogonal solution is to design less accurate and/or approximate algorithms such as statistical model checking that scale better and provide certain probabilistic guarantees about the correctness of verification. Another interesting direction is using counterexample-guided techniques to manage the size of the state space.

References

1. Ábrahám, E., Bartocci, E., Bonakdarpour, B., Dobe, O.: Parameter synthesis for probabilistic hyperproperties. In: Proc. of LPAR 2020: the 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 73, pp. 12–31. EasyChair (2020). https://doi.org/10.29007/37lf
2. Ábrahám, E., Bartocci, E., Bonakdarpour, B., Dobe, O.: Probabilistic hyperproperties with nondeterminism. In: Proc. of ATVA'20: the 18th Int. Symp. on Automated Technology for Verification. LNCS, vol. 12302, pp. 518–534. Springer (2020). https://doi.org/10.1007/978-3-030-59152-6
3. Ábrahám, E., Bonakdarpour, B.: HyperPCTL: A temporal logic for probabilistic hyperproperties. In: Proc. of QEST'18: the 15th International Conference on Quantitative Evaluation of Systems. LNCS, vol. 11024, pp. 20–35. Springer (2018). https://doi.org/10.1007/978-3-319-99154-2_2
4. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: Proc. of CSF'16: the IEEE 29th Computer Security Foundations. pp. 239–252. IEEE Computer Society (2016). https://doi.org/10.1109/CSF.2016.24
5. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
6. Bonakdarpour, B., Sánchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Proc. of ISoLA'18: the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Verification. LNCS, vol. 11245, pp. 8–27. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_2
7. Brett, N., Siddique, U., Bonakdarpour, B.: Rewriting-based runtime verification for alternation-free HyperLTL. In: Proc. of TACAS'17: the 23rd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 10206, pp. 77–93. Springer (2017). https://doi.org/10.1007/978-3-662-54580-5
8. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Proc. of CAV'19: the 31st Int. Conf. on Computer Aided Verification. LNCS, vol. 11561, pp. 121–139. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_7
9. Dimitrova, R., Finkbeiner, B., Torfah, H.: Probabilistic hyperproperties of Markov decision processes. In: Proc. of ATVA'20: the 18th Symposium on Automated Technology for Verification and Analysis. LNCS, vol. 12302, pp. 484–500. Springer (2020). https://doi.org/10.1007/978-3-030-59152-6
10. Dobe, O., Ábrahám, E., Bartocci, E., Bonakdarpour, B.: HyperProb: A model checker for probabilistic hyperproperties. In: Proc. of FM'21: the 24th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 13047, pp. 657–666. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6
11. Fallahi, N., Bonakdarpour, B., Tixeuil, S.: Rigorous performance evaluation of self-stabilization using probabilistic model checking. In: Proc. of SRDS'13: the 32nd IEEE Int. Conf. on Reliable Distributed Systems. pp. 153–162. IEEE Computer Society (2013). https://doi.org/10.1109/SRDS.2013.24
12. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: A runtime verification tool for temporal hyperproperties. In: Proc. of TACAS'18: the 24th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 10806, pp. 194–200. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_11

13. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. Formal Methods in System Design **54**(3), 336–363 (2019). https://doi.org/10.1007/s10703-019-00334-z
14. Finkbeiner, B., Hahn, C., Torfah, H.: Model checking quantitative hyperproperties. In: Proc. of CAV'18: the 30th Int. Conf. on Computer Aided Verification. LNCS, vol. 10981, pp. 144–163 (2018). https://doi.org/10.1007/978-3-319-96145-3_8
15. Finkbeiner, B., Müller, C., Seidl, H., Zalinescu, E.: Verifying security policies in multi-agent workflows with loops. In: Proc. of CCS'17: the 15th ACM Conf. on Computer and Communications Security (CCS). ACM (2017). https://doi.org/10.1145/3133956.3134080
16. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Proc. of CAV'15: the 27th Int. Conf. on Computer Aided Verification (CAV). LNCS, vol. 9206, pp. 30–48. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_3
17. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer (2011). https://doi.org/10.1007/978-3-642-21455-4
18. Hahn, C., Stenger, M., Tentrup, L.: Constraint-based monitoring of hyperproperties. In: Proc. of TACAS'19: the 25th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). vol. 11428, pp. 115–131. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1
19. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing **6**, 102–111 (1994). https://doi.org/10.1007/BF01211866
20. Herman, T.: Probabilistic self-stabilization. Information Processing Letters **35**(2), 63–67 (1990). https://doi.org/10.1016/0020-0190(90)90107-9
21. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self-stabilizing mutual exclusion. In: Proc. of PODC'90: the Ninth Annual ACM Symposium on Principles of Distributed Computing. pp. 119–131 (1990). https://doi.org/10.1145/93385.93409
22. Knuth, D., Yao, A.: Algorithms and Complexity: New Directions and Recent Results, chap. The complexity of nonuniform random number generation. Academic Press (1976)
23. Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic model checking. In: Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer (2007). https://doi.org/10.1007/978-3-540-72522-0
24. LARK. `https://lark-parser.readthedocs.io/`
25. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of TACAS'08. pp. 337–340 (2008)
26. STORMpy. `https://moves-rwth.github.io/stormpy/`
27. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Graybox monitoring of hyperproperties. In: Proc. of FM'19: the 23rd Int. Symp. on Formal Methods (FM). LNCS, vol. 11800, pp. 406–424. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8
28. Wang, Y., Nalluri, S., Bonakdarpour, B., Pajic, M.: Statistical model checking for hyperproperties. In: Proc. of CSF'21: the IEEE 34th Computer Security Foundations. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00009

29. Wang, Y., Zarei, M., Bonakdarpour, B., Pajic, M.: Statistical verification of hyperproperties for cyber-physical systems. ACM Trans. on Embedded Computing systems **18**(5s), 92:1–92:23 (2019). https://doi.org/10.1145/3358232