Introducing Asynchronicity to Probabilistic Hyperproperties

Lina Gerlach[1], Oyendrila Dobe[2], Erika Ábrahám[1], Ezio Bartocci[3], and Borzoo Bonakdarpour[2]

[1] RWTH Aachen University, Aachen, Germany,
{gerlach,abraham}@cs.rwth-aachen.de
[2] Michigan State University, East Lansing, MI, USA,
{dobeoyen,borzoo}@msu.edu
[3] Technische Universität Wien, Vienna, Austria.
ezio.bartocci@tuwien.ac.at

Abstract. Probabilistic hyperproperties express probabilistic relations between different executions of systems with uncertain behavior. HyperPCTL [3] allows to formalize such properties, where quantification over probabilistic schedulers resolves potential non-determinism. In this paper we propose an extension named AHyperPCTL to additionally introduce *asynchronicity* between the observed executions by quantifying over *stutter-schedulers*, which may randomly decide to delay scheduler decisions by idling. To our knowledge, this is the first asynchronous extension of a probabilistic branching-time hyperlogic.We show that AHyperPCTL can express interesting information-flow security policies, and propose a model checking algorithm for a decidable fragment.

1 Introduction

Consider the following simple multi-threaded program [29] consisting of two threads with a secret input h and a public output l:

$$th\colon \textbf{while } h > 0 \textbf{ do } \{h \leftarrow h - 1\};\ l \leftarrow 2 \quad \| \quad th'\colon l \leftarrow 1$$

Assuming that this program is executed under a probabilistic scheduler, the probability of observing $l = 1$ decreases for increasing initial values of h. Hence, this program does not satisfy scheduler-specific probabilistic observational determinism (SSPOD) [27], which requires that no information about the private data is leaked through the publicly visible data, for any scheduling of the threads. In fact, the scheduler is creating a probabilistic side channel that leaks the value of the secret. Probabilistic hyperlogics such as HyperPCTL [2,3,20] and PHL [17] are able to express and verify requirements such as SSPOD.

Interestingly, there is a way to mitigate this side channel similar to the padding mechanism that counters timing side channels. In the above example, for any two executions of the program under the same scheduler with different initial h, we can find *stuttering variations* of the program such that the probability of reaching any specific final value of l is the same for both executions.

For example, for two different values of h, say h_1 and h_2, where $h_1 < h_2$, letting thread th' initially stutter $(h_2 - h_1)$ times (i.e., repeating the current state) in the execution starting with h_1 will equalize the probability of reaching $l = 1$.

While there have been efforts to incorporate stuttering semantics in non-probabilistic logics (e.g., A-HLTL [8]), in the probabilistic setting, neither HyperPCTL nor PHL allow reasoning about stuttering behaviors, i.e., their semantics are "synchronous" in the sense that all computation trees are evaluated in lock-step. In this paper, we propose an asynchronous extension of HyperPCTL that allows to reason about stuttering computations and whether we can find stuttering variations of programs such that a probabilistic hyperproperty is satisfied.

Related Work HyperPCTL [3] was the first logic for specifying probabilistic hyperproperties over DTMCs, by providing state-based quantifiers over the computation trees of the DTMC. This logic was further extended [1, 2, 18–20] with the possibility to specify quantifiers over schedulers for model checking Markov decision processes (MDPs). The probabilistic hyperlogic PHL [17] can also handle analysis of MDPs. In general, the (exact) model checking problem for both HyperPCTL and PHL is undecidable unless we restrict the class of schedulers or we rely on some approximating methods. HyperPCTL* [30, 31] extends PCTL* [5] with quantifiers over execution paths and is employed in statistical model checking. All three logics are synchronous and lock-step. To the best of our knowledge, our work is the first to consider asynchronicity in the probabilistic setting.

In the non-probabilistic setting, asynchronicity has already been studied [7, 9, 10, 12, 13, 23]. In [7], the authors study the expressivity of HyperLTL [15] showing the impossibility to express the "two-state local independence" asynchronous hyperproperty, where information flow is allowed only after a change of state (for example in the case of declassification [28]). To cope with this limitation, several asynchronous extensions of HyperLTL have been proposed.

For example, *Asynchronous HyperLTL* [10] extends HyperLTL with quantification over "trajectories" that enable the alignment of execution traces from different runs. *Stuttering HyperLTL* [12] relates only stuttering traces where two consecutive observations are different. *Context HyperLTL* [12] instead allows to combine synchronous and asynchronous hyperproperties. All three logics are in general undecidable, but there are useful decidable fragments that can be model-checked. The expressiveness of these logics has been compared in [13].

Contributions Our main contribution is a new logic, called AHyperPCTL, which is an asynchronous extension of HyperPCTL and allows to reason about probabilistic relations between stuttering variations of probabilistic and potentially non-deterministic systems. To our knowledge, this is the first asynchronous extension of a probabilistic branching-time hyperlogic. Our goal is to associate several executions with independent stuttering variations of the same program and compare them. We implement this by extending HyperPCTL with quantification over *stutter-schedulers*, which specify when the program should stutter.

We show that AHyperPCTL is useful to express whether information leaks can be avoided via suitable stuttering. In the context of our introductory example,

the following AHyperPCTL formula expresses SSPOD under stuttering:

$$\forall\hat{\sigma}.\ \forall\hat{s}(\hat{\sigma}).\ \forall\hat{s}'(\hat{\sigma}).\ \exists\hat{\tau}(\hat{s}).\ \exists\hat{\tau}'(\hat{s}').$$
$$(h_{\hat{\tau}}{\neq}h_{\hat{\tau}'} \wedge \mathit{init}_{\hat{\tau}} \wedge \mathit{init}_{\hat{\tau}'}) \Rightarrow \big(\textstyle\bigwedge_{k\in\{1,2\}} \mathbb{P}\big(\Diamond(l{=}k)_{\hat{\tau}}\big) = \mathbb{P}\big(\Diamond(l{=}k)_{\hat{\tau}'}\big)\big),$$

where $\hat{\sigma}$ represents a probabilistic scheduler that specifies which thread is allowed to execute in which program state, $\hat{s}$ and $\hat{s}'$ represent initial states, and $\hat{\tau}$ and $\hat{\tau}'$ are stutter-scheduler variables for the computation trees rooted at $\hat{s}$ and $\hat{s}'$ under the scheduler $\hat{\sigma}$. This formula specifies that under any probabilistic scheduling $\hat{\sigma}$ of the two threads, if we consider two computation trees starting in states $\hat{s}$ and $\hat{s}'$ with different values for the secret variable h, there should exist stutterings for the two experiments such that the probabilities of observing any specific final value of l are the same for both.

We propose a model checking algorithm for AHyperPCTL under restrictions on the classes of schedulers and stutter-schedulers. Our method generates a logical encoding of the problem in real arithmetic and uses a satisfiability modulo theories (SMT) solver, namely Z3 [25], to determine the truth of the input statement. We experimentally demonstrate that the model checking problem for asynchronous probabilistic hyperproperties is a computationally highly complex synthesis problem at two levels: both for synthesizing scheduler policies and for synthesizing stutter schedulers. This poses serious problems for model checking: our current implementation does not scale beyond a few states. We discuss some insights about this complexity and suggest possible future directions.

Organization We discuss preliminary concepts in Sec. 2 and introduce AHyperPCTL in Sec. 3. We dedicate Sec. 4 to applications and Sec. 5 to our algorithm. We discuss results of our prototype implementation in Sec. 6. We conclude in Sec. 7 with a summary and future work.

2 Preliminaries

We denote the real (non-negative real) numbers by $\mathbb{R}$ ($\mathbb{R}_{\geq 0}$), and the natural numbers including (excluding) 0 by $\mathbb{N}$ ($\mathbb{N}_{>0}$). For any $n \in \mathbb{N}$, we define $[n]$ to be the set $\{0,\dots,n{-}1\}$. We use $()$ to denote the empty tuple and $\circ$ for concatenation.

Definition 1. *A* discrete-time Markov chain (DTMC) *is a tuple* $\mathcal{D}{=}(S, \mathsf{AP}, L, \mathbf{P})$ *where (1)* S *is a non-empty finite set of* states*, (2)* AP *is a set of* atomic propositions*, (3)* $L\colon S \to 2^{AP}$ *is a* labeling function *and (4)* $\mathbf{P}\colon S \times S \to [0,1]$ *is a* transition probability function *such that* $\sum_{s'\in S} \mathbf{P}(s,s') = 1$ *for all* $s \in S$*.*

An *(infinite) path* is a sequence $s_0 s_1 s_2 \ldots \in S^\omega$ of states with $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. Let $\mathit{Paths}_s^{\mathcal{D}}$ denote the set of all paths of $\mathcal{D}$ starting in $s \in S$, and $\mathit{fPaths}_s^{\mathcal{D}}$ denote the set of all non-empty finite prefixes of paths from $\mathit{Paths}_s^{\mathcal{D}}$, which we call *finite paths*. For a finite path $\pi = s_0 \ldots s_k \in \mathit{fPaths}_{s_0}^{\mathcal{D}}$, $k \geq 0$, we define $|\pi| = k$. A state $t \in S$ is *reachable* from $s \in S$ if there exists a finite path

in $fPaths_s^{\mathcal{D}}$ that ends in t. The *cylinder set* $Cyl^{\mathcal{D}}(\pi)$ of a finite path π is the set of all infinite paths with π as a prefix. The *probability space* for $\mathcal{D}$ and $s \in S$ is

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

where the *probability* of the cylinder set of $\pi \in fPaths_s^{\mathcal{D}}$ is $\Pr_s^{\mathcal{D}}(Cyl^{\mathcal{D}}(\pi)) = \Pi_{i=1}^{|\pi|} \mathbf{P}(\pi_{i-1}, \pi_i)$. These concepts have been discussed in detail in [5].

Markov decision processes extend DTMCs to allow the modeling of environment interaction or user input in the form of non-determinism.

Definition 2. *A* Markov decision process (MDP) *is defined as a tuple* $\mathcal{M} = (S, \mathsf{AP}, L, Act, \mathbf{P})$*, where (1)* S *is a non-empty finite set of* states*, (2)* AP *is a set of* atomic propositions*, (3)* $L \colon S \to 2^{\mathsf{AP}}$ *is a* labeling function*, (4)* Act *is a non-empty finite set of* actions*, (5)* $\mathbf{P} \colon S \times Act \times S \to [0, 1]$ *is a* transition probability function *such that for all* $s \in S$ *the set of its* enabled actions

$$Act(s) = \left\{ \alpha \in Act \mid \sum_{s' \in S} \mathbf{P}(s, \alpha, s') = 1 \right\}$$

is non-empty and $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') = 0$ *for all* $\alpha \in Act \setminus Act(s)$*.*

Let $\mathbb{M}$ be the set of all MDPs. For every execution step of an MDP, a *scheduler* resolves the non-determinism by selecting an enabled action to be executed.

Definition 3. *For an MDP* $\mathcal{M} = (S, \mathsf{AP}, L, Act, \mathbf{P})$*, a* scheduler *is a tuple* $\sigma = (Q, mode, init, act)$*, where (1)* Q *is a non-empty countable set of* modes*, (2)* $mode \colon Q \times S \to Q$ *is a* mode transition function*, (3)* $init \colon S \to Q$ *selects the* starting mode $init(s)$ *for each state of* $s \in S$*, and (4)* $act \colon Q \times S \times Act \to [0, 1]$ *is a function with* $\sum_{\alpha \in Act(s)} act(q, s, \alpha) = 1$ *and* $\sum_{\alpha \in Act \setminus Act(s)} act(q, s, \alpha) = 0$ *for all* $s \in S$ *and* $q \in Q$*.*

We use $\Sigma^{\mathcal{M}}$ to denote the set of all schedulers for an MDP $\mathcal{M}$. A scheduler is called *finite-memory* if Q is finite, *memoryless* if Q is a singleton, and *deterministic* if $act(q, s, \alpha) \in \{0, 1\}$ for all $(q, s, \alpha) \in Q \times S \times Act$. If a scheduler is memoryless, we sometimes omit its only mode.

3 Asynchronous HyperPCTL

Probabilistic hyperproperties specify probabilistic relations between different executions of one or several probabilistic models. In previous work, we introduced HyperPCTL [3] to reason over non-determinism [20] and rewards [19] for *synchronous* executions, i.e., where all executions make their steps simultaneously. In this work, we propose an extension to reason about *asynchronous* executions, where some of the executions may also *stutter* (i.e., stay in the same state without observable changes) while others execute.

This is useful if, for example, the duration of some computations depend on some secret input and we thus might wish to make the respective duration unobservable. A typical application area are multi-threaded programs, like the one presented in Section 1, where we want to relate the executions of the different

threads. If there is a single processor available, each execution step allows one of the threads to execute, while the others idle. The executing thread, however, might also decide to stutter in order to hide its execution duration. To be able to formalize such behavior, the decision whether an execution stutters or not must depend not only on the history but also on the chosen action (in our example, corresponding to which of the threads may execute).

In this section, we first introduce a novel scheduler concept that supports stuttering, followed by the extension of HyperPCTL that we henceforth refer to as AHyperPCTL. To improve readability, we assume that all executions run in the same MDP; an extension to different MDPs is a bit technical but straightforward.

3.1 Stutter Schedulers

We define a *stutter-scheduler* as an additional type of scheduler that only distinguishes between stuttering, represented by ε, or proceeding, represented by $\overline{\varepsilon}$, for every state $s \in S$ and action $\alpha \in Act$.

Definition 4. *A* stutter-scheduler *for an MDP* $\mathcal{M} = (S, \mathsf{AP}, L, Act, \mathbf{P})$ *is a tuple* $\tau = (Q^\varepsilon, mode^\varepsilon, init^\varepsilon, act^\varepsilon)$ *where (1)* Q^ε *is a non-empty countable set of* modes*, (2)* $mode^\varepsilon \colon Q^\varepsilon \times S \times Act \to Q^\varepsilon$ *is a* mode transition function*, (3)* $init^\varepsilon \colon S \to Q^\varepsilon$ *is a function selecting a* starting mode $init(s)$ *for each state* $s \in S$ *and (4)* $act^\varepsilon \colon Q^\varepsilon \times S \times Act \times \{\varepsilon, \overline{\varepsilon}\} \to [0,1]$ *is a function with*

$$act^\varepsilon(q^\varepsilon, s, \alpha, \varepsilon) + act^\varepsilon(q^\varepsilon, s, \alpha, \overline{\varepsilon}) = 1$$

for all $(q^\varepsilon, s, \alpha) \in Q^\varepsilon \times S \times Act$.

We use $\mathcal{T}^\mathcal{M}$ to denote the set of all stutter-schedulers for an MDP $\mathcal{M}$. When reasoning about asynchronicity, we consider an MDP $\mathcal{M}$ in the context of a scheduler and a stutter-scheduler for $\mathcal{M}$. At each state, first the scheduler chooses an action α, followed by a decision of the stutter-scheduler whether to execute α or to stutter (i.e., stay in the current state). Thus, a stutter-scheduler makes its decisions based on not only its mode and the MDP state, but also depending on the action chosen by the scheduler.

Definition 5. *For an MDP* $\mathcal{M} = (S, \mathsf{AP}, L, Act, \mathbf{P})$*, a scheduler* σ *for* $\mathcal{M}$ *and a stutter-scheduler* τ *for* $\mathcal{M}$*, the* DTMC induced by σ and τ *is defined as* $\mathcal{M}^{\sigma,\tau} = (S^{\sigma,\tau}, \mathsf{AP}, L^{\sigma,\tau}, \mathbf{P}^{\sigma,\tau})$*, where* $S^{\sigma,\tau} = Q \times Q^\varepsilon \times S$*,* $L^{\sigma,\tau}(q, q^\varepsilon, s) = L(s)$ *and*

$$\mathbf{P}^{\sigma,\tau}((q,q^\varepsilon,s),(q',q^{\varepsilon\prime},s')) = \begin{cases} stut & \text{if } q' = q \neq mode(q,s) \wedge s' = s \\ cont & \text{if } q' = mode(q,s) \wedge (q' \neq q \vee s' \neq s) \\ stut + cont & \text{if } q' = q = mode(q,s) \wedge s' = s \\ 0 & \text{otherwise} \end{cases}$$

with $stut = \sum_{\alpha \in Act, mode^\varepsilon(q^\varepsilon,s,\alpha)=q^{\varepsilon\prime}} act(q,s,\alpha) \cdot act^\varepsilon(q^\varepsilon, s, \alpha, \varepsilon)$

and $cont = \sum_{\alpha \in Act, mode^\varepsilon(q^\varepsilon,s,\alpha)=q^{\varepsilon\prime}} act(q,s,\alpha) \cdot act^\varepsilon(q^\varepsilon, s, \alpha, \overline{\varepsilon}) \cdot \mathbf{P}(s,\alpha,s')$.

The properties *finite-memory*, *memoryless*, and *deterministic* for stutter schedulers are defined analogously as for schedulers. A stutter-scheduler τ for an MDP $\mathcal{M}$ is *fair* for a scheduler $\sigma \in \Sigma^{\mathcal{M}}$ if the probability of taking infinitely many consecutive stuttering steps is 0. The different executions, whose relations we want to analyze, will be evaluated in the *composition* of the induced models. The composition we use is the standard product of DTMCs with the only difference that we annotate atomic propositions with an index, indicating the execution in which they appear.

Definition 6. *For $n \in \mathbb{N}$ and DTMCs $\mathcal{D}_1, \ldots, \mathcal{D}_n$ with $\mathcal{D}_i = (S_i, \mathsf{AP}_i, L_i, \mathbf{P}_i)$ for $i = 1, \ldots, n$, we define the* composition *$\mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ to be the DTMC $\mathcal{D} = (S, \mathsf{AP}, L, \mathbf{P})$ with $S = S_1 \times \ldots \times S_n$, $\mathsf{AP} = \cup_{i=1}^{n} \{a_i \mid a \in \mathsf{AP}_i\}$, $L(s_1, \ldots, s_n) = \cup_{i=1}^{n} \{a_i \mid a \in L_i(s_i)\}$ and $P((s_1, \ldots, s_n), (s'_1, \ldots, s'_n)) = \Pi_{i=1}^{n} P_i(s_i, s'_i)$.*

Definition 7. *For an MDP $\mathcal{M}$, $n \in \mathbb{N}_{>0}$, a tuple $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n) \in (\Sigma^{\mathcal{M}})^n$ of schedulers, and a tuple $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_n) \in (\mathcal{T}^{\mathcal{M}})^n$ of stutter-schedulers, we define the* induced DTMC *$\mathcal{M}^{\boldsymbol{\sigma}, \boldsymbol{\tau}} = \mathcal{M}^{\sigma_1, \tau_1} \times \ldots \times \mathcal{M}^{\sigma_n, \tau_n}$.*

Later we will make use of *counting* stutter-schedulers. These are deterministic bounded-memory stutter-schedulers which specify for each state $s \in S$ and action $\alpha \in Act(s)$ a stuttering duration $j_{s,\alpha}$. Intuitively, $j_{s,\alpha}$ determines how many successive stutter-steps need to be made in state s before α can be executed.

Definition 8. *An m-*bounded counting *stutter-scheduler for an MDP $\mathcal{M} = (S, \mathsf{AP}, L, Act, \mathbf{P})$ and $m \in \mathbb{N}_{>0}$ is a stutter-scheduler $\tau = ([m], mode^{\varepsilon}, init^{\varepsilon}, act^{\varepsilon})$ such that for all $s \in S$ and $\alpha \in Act(s)$ there exists $j_{s,\alpha} \in [m]$ with (1) $init^{\varepsilon}(s)$=0 and (2) for each $j \in [m]$, if $j < j_{s,\alpha}$ then $mode^{\varepsilon}(j, s, \alpha)$=$j$+1 and $act^{\varepsilon}(j, s, \alpha, \varepsilon) = 1$, and otherwise (if $j \geq j_{s,\alpha}$) $mode^{\varepsilon}(j, s, \alpha) = 0$ and $act^{\varepsilon}(j, s, \alpha, \overline{\varepsilon}) = 1$.*

Example 1. Consider the MDP $\mathcal{M}$ from Figure 1 as well as the DTMC $\mathcal{M}^{\sigma,\tau}$ induced by a probabilistic memoryless scheduler σ with $\sigma(s_0, \alpha) = p$, $\sigma(s_0, \beta) = 1-p$ for some $p \in [0, 1]$ and a 3-counting stutter-scheduler τ on $\mathcal{M}$ with $j_{s_0,\alpha} = 2$, $j_{s_0,\beta} = 0$. The modes $q \in [3]$ of τ store how many times we have stuttered since actually executing the last action. For each state (s, j) of $\mathcal{M}^{\sigma,\tau}$, first σ chooses an action $\alpha \in Act(s)$ probabilistically, and then j is compared with the stuttering duration $j_{s,\alpha}$ stipulated by τ. If we choose β in state $(s_0, 0)$, then we move to a β-successor of s_0. However, if we choose α, then we move to the state $(s_0, 1)$ and then choose an action again. If we choose β at $(s_0, 1)$, then we move to a β-successor of s_0, but if we choose α then we move to $(s_0, 2)$. In particular, choosing α at $(s_0, 0)$ does not mean that we stutter twice in s_0. We stutter twice only if we also choose α in $(s_0, 1)$.

3.2 Syntax

To formalize relations of different executions, we begin an AHyperPCTL formula as in HyperPCTL [20] by first quantifying over the possible schedulers and then over the states of the MDP in which the respective execution under the chosen

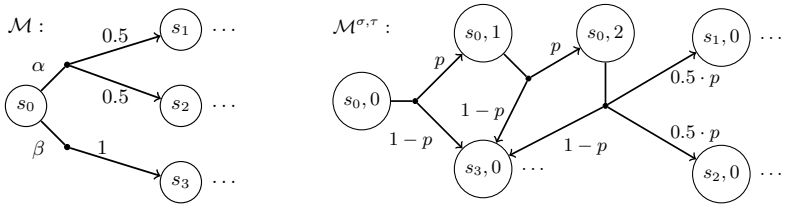

Fig. 1: The interplay of a probabilistic memoryless scheduler σ and a counting stutter-scheduler τ from Ex. 1. (The mode of σ is omitted.)

scheduler starts. In contrast to [20], we now additionally quantify over stutter-schedulers in dependence on the chosen schedulers and initial states. Hence, the *non-quantified* part of an AHyperPCTL formula is evaluated on the DTMC(s) induced by not only the schedulers but also the stutter-schedulers, in accordance with Def. 5. Formally, we inductively define AHyperPCTL scheduler-quantified formulas as follows:

$$
\begin{array}{lll}
\textit{scheduler} - \textit{quantified}: & \varphi^{sch} & ::= \forall \hat{\sigma}.\varphi^{sch} \mid \exists \hat{\sigma}.\varphi^{sch} \mid \varphi^{s} \\
\textit{state} - \textit{quantified}: & \varphi^{s} & ::= \forall \hat{s}(\hat{\sigma}).\varphi^{s} \mid \exists \hat{s}(\hat{\sigma}).\varphi^{s} \mid \varphi^{st} \\
\textit{stutter} - \textit{quantified}: & \varphi^{st} & ::= \forall \hat{\tau}(\hat{s}).\varphi^{st} \mid \exists \hat{\tau}(\hat{s}).\varphi^{st} \mid \varphi^{nq} \\
\textit{non} - \textit{quantified}: & \varphi^{nq} & ::= \texttt{true} \mid a_{\hat{\tau}} \mid \varphi^{nq} \wedge \varphi^{nq} \mid \neg\varphi^{nq} \mid \varphi^{pr} \sim \varphi^{pr} \\
\textit{probability expression}: & \varphi^{pr} & ::= \mathbb{P}(\varphi^{path}) \mid f(\varphi_1^{pr}, \ldots, \varphi_k^{pr}) \\
\textit{path formula}: & \varphi^{path} & ::= \bigcirc \varphi^{nq} \mid \varphi^{nq}\, \mathcal{U}\, \varphi^{nq}
\end{array}
$$

where $\hat{\sigma}$ is a *scheduler variable* from an infinite set $\hat{\Sigma}$, $\hat{s}$ is a *state variable* from an infinite set $\hat{S}$, $\hat{\tau}$ is a *stutter-scheduler variable* from an infinite set $\hat{\mathcal{T}}$, $a \in \mathsf{AP}$ is an atomic proposition, $\sim \in \{\leq, <, =, \neq, >, \geq\}$, and $f\colon [0,1]^k \to \mathbb{R}$ is a k-ary arithmetic operator over probabilities, where a constant c is viewed as a 0-ary function. $\mathbb{P}$ refers to the probability operator and '$\bigcirc$', '$\mathcal{U}$' refer to the temporal operators 'next' and 'until', respectively.

An AHyperPCTL scheduler-quantified formula φ^{sch} is *well-formed* if each occurrence of any $a_{\hat{\tau}}$ for $\hat{\tau} \in \hat{\mathcal{T}}$ is in the scope of a stutter quantifier for $\hat{\tau}(\hat{s})$ for some $\hat{s} \in \hat{S}$, any quantifier for $\hat{\tau}(\hat{s})$ is in the scope of a *state quantifier* for $\hat{s}(\hat{\sigma})$ for some $\hat{\sigma} \in \hat{\Sigma}$, and any quantifier for $\hat{s}(\hat{\sigma})$ is in the scope of a *scheduler quantifier* for $\hat{\sigma}$. AHyperPCTL *formulas* are well-formed AHyperPCTL scheduler-quantified formulas, where we additionally allow standard syntactic sugar: $\texttt{false} = \neg\texttt{true}$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 = \neg(\varphi_1 \wedge \neg\varphi_2)$, $\diamondsuit\varphi = \texttt{true}\, \mathcal{U}\, \varphi$, and $\mathbb{P}(\square\varphi) = 1 - \mathbb{P}(\diamondsuit\neg\varphi)$.

Example 2. The well-formed AHyperPCTL formula

$$\exists \hat{\sigma}.\ \forall \hat{s}(\hat{\sigma}).\ \forall \hat{s}'(\hat{\sigma}).\ \exists \hat{\tau}(\hat{s}).\ \exists \hat{\tau}'(\hat{s}').\ \left(init_{\hat{\tau}} \wedge init_{\hat{\tau}'}\right) \Rightarrow \left(\mathbb{P}(\Diamond a_{\hat{\tau}}) = \mathbb{P}(\Diamond a_{\hat{\tau}'})\right)$$

states that there exists an assignment for $\hat{\sigma}$, such that, if we start two independent *experiments* from any state assignment to $\hat{s}$ and $\hat{s}'$, there exist independent possible stutter-schedulers for the two experiments, under which the probability of reaching a state labeled a is equal, provided the initial states of the experiments are labeled *init*. Further examples will be provided in Sec. 4.

We restrict ourselves to quantifying first over schedulers, then over states, and finally over stutter-schedulers. This choice is a balance between the expressivity required in our applications and understandable syntax and semantics. Note that different state variables can share the same scheduler, but they cannot share the same stutter-scheduler. Further, several different quantified stutter-schedulers in a formula are not allowed to depend on each other.

3.3 Semantics

The semantic judgement rules for AHyperPCTL closely mirror the rules for HyperPCTL [2]. AHyperPCTL state formulas are evaluated in the context of an MDP $\mathcal{M}$, a sequence $\boldsymbol{\sigma} \in (\Sigma^{\mathcal{M}})^n$ of schedulers, a sequence $\boldsymbol{\tau} \in (\mathcal{T}^{\mathcal{M}})^n$ of stutter-schedulers and a sequence $\boldsymbol{s} \in S^{\boldsymbol{\sigma},\boldsymbol{\tau}}$ of $\mathcal{M}^{\boldsymbol{\sigma},\boldsymbol{\tau}}$-states. The length n of these tuples corresponds to the number of stutter-schedulers in the given formula, which determines how many experiments run in parallel. The elements of these tuples are instantiations of the corresponding variables in the formula. We assume the stutter-schedulers to be fair and the variables used to refer to each of these quantifiers in the formula to be unique to avoid ambiguity. In the following, we use $\mathbb{Q}$ to refer to quantifiers $\{\forall, \exists\}$. We recursively evaluate the formula by instantiating the quantifier variables with concrete schedulers, states, and stutter-schedulers, and store them in sequences $\boldsymbol{\sigma}, \boldsymbol{s}, \boldsymbol{\tau}$. We begin by initializing each of these sequences as empty. An MDP $\mathcal{M} = (S, \mathsf{AP}, L, Act, \mathbf{P})$ satisfies an AHyperPCTL formula φ, denoted by $\mathcal{M} \models \varphi$, iff $\mathcal{M}, (), (), () \models \varphi$.

When instantiating a scheduler quantifier $\mathbb{Q}\hat{\sigma}.\varphi$ by a scheduler σ, we syntactically replace all occurrences of $\hat{\sigma}$ in φ by σ and denote the result by $\varphi[\hat{\sigma} \rightsquigarrow \sigma]$[4]. The instantiation of a state quantifier $\mathbb{Q}\hat{s}(\hat{\sigma}).\varphi$ works similarly but it also remembers the respective scheduler: $\varphi[\hat{s} \rightsquigarrow s_\sigma]$ denotes the result of syntactically replacing all occurrences of $\hat{s}$ in φ by s_σ. Finally, for instantiating the n^{th} stutter-scheduler quantifier $\mathbb{Q}\hat{\tau}(s_\sigma).\varphi$, we replace all occurrences of $a_{\hat{\tau}}$ by a_n and denote the result by $\varphi[\hat{\tau} \overset{n}{\rightsquigarrow} \tau]$. The semantics judgement rules for quantified and non-quantified state formulas, as well as probability expressions are defined as

[4] Note that we substitute a syntactic element with a semantic object in order to reduce notation; alternatively one could store respective mappings in the context.

follows:

$$
\begin{array}{lll}
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \forall \hat{\sigma}.\ \varphi & \textit{iff} & \forall \sigma \in \Sigma^{\mathcal{M}}.\ \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \varphi[\hat{\sigma} \rightsquigarrow \sigma] \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \exists \hat{\sigma}.\ \varphi & \textit{iff} & \exists \sigma \in \Sigma^{\mathcal{M}}.\ \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \varphi[\hat{\sigma} \rightsquigarrow \sigma] \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \forall \hat{s}(\sigma).\ \varphi & \textit{iff} & \forall s \in S.\ \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \varphi[\hat{s} \rightsquigarrow s_\sigma] \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \exists \hat{s}(\sigma).\ \varphi & \textit{iff} & \exists s \in S.\ \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \varphi[\hat{s} \rightsquigarrow s_\sigma] \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \forall \hat{\tau}(s_\sigma).\ \varphi & \textit{iff} & \forall \tau \in \mathcal{T}^{\mathcal{M}}.\ \mathcal{M},\boldsymbol{\sigma} \circ \sigma,\boldsymbol{s} \circ (\mathit{init}(s), \mathit{init}^{\varepsilon}(s), s), \\
 & & \qquad \boldsymbol{\tau} \circ \tau \models \varphi[\hat{\tau} \overset{n}{\rightsquigarrow} \tau] \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \exists \hat{\tau}(s_\sigma).\ \varphi & \textit{iff} & \exists \tau \in \mathcal{T}^{\mathcal{M}}.\ \mathcal{M},\boldsymbol{\sigma} \circ \sigma,\boldsymbol{s} \circ (\mathit{init}(s), \mathit{init}^{\varepsilon}(s), s), \\
 & & \qquad \boldsymbol{\tau} \circ \tau \models \varphi[\hat{\tau} \overset{n}{\rightsquigarrow} \tau] \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \texttt{true} & & \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models a_i & \textit{iff} & a_i \in L^{\boldsymbol{\sigma},\boldsymbol{\tau}}(\boldsymbol{s}) \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \varphi_1 \text{ and } \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \varphi_2 \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \neg\varphi & \textit{iff} & \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \not\models \varphi \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau} \models \varphi_1^{pr} < \varphi_2^{pr} & \textit{iff} & [\![\varphi_1^{pr}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau}} < [\![\varphi_2^{pr}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau}} \\
[\![\mathbb{P}(\varphi^{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau}} & = & Pr_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma},\boldsymbol{\tau}}}(\{\pi \in Paths_{\boldsymbol{s}}^{\mathcal{M}^{\boldsymbol{\sigma},\boldsymbol{\tau}}} \mid \mathcal{M},\boldsymbol{\sigma},\pi,\boldsymbol{\tau} \models \varphi^{path}\}) \\
[\![f(\varphi_1^{pr},\ldots,\varphi_k^{pr})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau}} & = & f([\![\varphi_1^{pr}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau}},\ldots,[\![\varphi_k^{pr}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s},\boldsymbol{\tau}})
\end{array}
$$

where the tuples $\boldsymbol{\sigma}$, $\boldsymbol{s}$ and $\boldsymbol{\tau}$ are of length $n-1$. The semantics of path formulas is defined as follows for a path $\pi = \boldsymbol{s}_0\boldsymbol{s}_1\ldots$ of $\mathcal{M}^{\boldsymbol{\sigma},\boldsymbol{\tau}}$ with $\boldsymbol{s}_i \in S^{\sigma_1,\tau_1} \times \ldots \times S^{\sigma_n,\tau_n}$:

$$
\begin{array}{lll}
\mathcal{M},\boldsymbol{\sigma},\pi,\boldsymbol{\tau} \models \bigcirc\varphi & \textit{iff} & \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}_1,\boldsymbol{\tau} \models \varphi \\
\mathcal{M},\boldsymbol{\sigma},\pi,\boldsymbol{\tau} \models \varphi_1\,\mathcal{U}\,\varphi_2 & \textit{iff} & \exists j \geq 0.\ \big(\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}_j,\boldsymbol{\tau} \models \varphi_2 \wedge \\
 & & \qquad \forall i \in [0,j).\ \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}_i,\boldsymbol{\tau} \models \varphi_1\big)
\end{array}
$$

Lemma 1. *AHyperPCTL is strictly more expressive than HyperPCTL.*

Proof (Sketch). For every MDP $\mathcal{M}$ and HyperPCTL formula φ, we can construct an MDP $\mathcal{M}'$ and an AHyperPCTL formula φ' such that $\mathcal{M} \models_{\mathsf{HyperPCTL}} \varphi$ iff $\mathcal{M}' \models_{\mathsf{AHyperPCTL}} \varphi'$. The MDP $\mathcal{M}'$ is constructed from $\mathcal{M}$ by transforming each self-loop to a two-state-loop and then adding a unique label a_s to each state s. For this MDP, we define a formula $\mathit{trivial}_{\hat{\tau}_1,\ldots,\hat{\tau}_m}$ that checks whether the given stutter-schedulers are trivial by requiring that the probability of seeing the same state label a_s in the current and in the next step must always be 0. We construct φ' by adding a universal stutter-quantifier for each state quantifier and requiring that if these stutter-schedulers are all trivial, then the original non-quantified formula must hold.

AHyperPCTL is thus at least as expressive as HyperPCTL, and since HyperPCTL cannot express stutter quantification, AHyperPCTL is strictly more expressive.

Hence, since the model checking problem for HyperPCTL is already undecidable [3], it follows that AHyperPCTL model checking is undecidable as well.

Theorem 1. *The AHyperPCTL model checking problem is undecidable.*

4 Applications of AHyperPCTL

ACDB Consider the code snippet [22] in Fig. 2, where two threads synchronize across a critical region realized by the `await semaphore` command. Different interleavings of the threads can yield different sequences of observable outputs (i.e., permutations of `abcd`). Assume this program is executed according to a probabilistic scheduler. Since the behavior of thread `T2` depends on a secret input `h`, under synchronous semantics, the program leaks information about the secret input: the probability of observing output sequence `acdb` is 0 if `h` = 0 and non-zero for `h` = 1. However, stuttering after line 8 until `b` is printed would prevent an information leak. The following AHyperPCTL formula expresses a variation of SSPOD, requiring that the probability of observing any specific output should be the same regardless of the value of `h`:

```
1   Thread T1(){
2    await semaphore{
3     print('a');
4     v = v+1;
5     print('b');}
6   }
7   Thread T2(){
8    print('c');
9    if h=1{
10    await semaphore{
11     v = v+2;}}
12   print('d');
13  }
```

Fig. 2: Information leak.

$$\forall\hat{\sigma}.\ \forall\hat{s}(\hat{\sigma}).\ \forall\hat{s}'(\hat{\sigma}).\ \exists\hat{\tau}(\hat{s}).\ \exists\hat{\tau}'(\hat{s}').$$
$$(h_{\hat{\tau}} \neq h_{\hat{\tau}'} \wedge init_{\hat{\tau}} \wedge init_{\hat{\tau}'}) \Rightarrow \left(\mathbb{P}\left(\square \bigwedge\nolimits_{a\in obs} \mathbb{P}(\bigcirc a_{\hat{\tau}}) = \mathbb{P}(\bigcirc a_{\hat{\tau}'})\right) = 1\right)$$

Side-Channel Timing Leaks are a kind of information leak where an attacker can infer the approximate value of the secret input based on the difference in execution time for different inputs to the algorithm. Stuttering could hide these differences. Consider the code snippet in Fig. 3 representing the modular exponentiation algorithm, which is part of the RSA public-key encryption protocol. It computes the value of a^b mod n where `a` (integer) is the plaintext and `b` (integer) is the encryption key. In [2], we verified that we can notice the timing difference using a synchronous logic. We formalize in AHyperPCTL that for *any* possible scheduling of the two threads there exists possible stuttering that prevents the timing leak:

```
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  t = new Thread(mexp());
12  j = 0; m = 2 * k;
13  while (j < m & !t.stop){
14   j++;}
```

Fig. 3: Modular exponentiation.

$$\forall\hat{\sigma}.\ \forall\hat{s}(\hat{\sigma}).\ \forall\hat{s}'(\hat{\sigma}).\ \exists\hat{\tau}(\hat{s}).\exists\hat{\tau}'(\hat{s}').$$
$$(h_{\hat{\tau}} \neq h_{\hat{\tau}'} \wedge init_{\hat{\tau}} \wedge init_{\hat{\tau}'}) \Rightarrow (\textstyle\bigwedge_{l=0}^{m} \mathbb{P}(\Diamond(j=l)_{\hat{\tau}}) = \mathbb{P}(\Diamond(j=l)_{\hat{\tau}'}))$$

5 Model Checking AHyperPCTL

Due to general undecidability, we propose a model checking algorithm for a practically useful semantical fragment of AHyperPCTL: (1) we restrict scheduler quantification to probabilistic memoryless schedulers such that the same probabilistic decisions are made in states with identical enabled action sets, i.e., if

Algorithm 1: Main SMT encoding algorithm

Input: $\mathcal{M} = (S, Act, \mathbf{P}, \mathsf{AP}, L)$: MDP; m: Memory size for the stutter-schedulers; $\exists \hat{\sigma} Q_1 \hat{s}_1(\hat{\sigma}) \dots Q_l \hat{s}_l(\hat{\sigma}) \exists \hat{\tau}_1(\hat{s}_{k_1}) \dots \exists \hat{\tau}_n(\hat{s}_{k_n}).\varphi^{nq}$: AHyperPCTL formula.

Output: Whether $\mathcal{M}$ satisfies the input formula.

1 **Function** `Main`$(\mathcal{M}, m, \exists \hat{\sigma} Q_1 \hat{s}_1(\hat{\sigma}) \dots Q_l \hat{s}_l(\hat{\sigma}) \exists \hat{\tau}_1(\hat{s}_{k_1}) \dots \exists \hat{\tau}_n(\hat{s}_{k_n}).\varphi^{nq})$

2 $\quad \varphi_{sch} := \bigwedge_{\emptyset \neq A \subseteq Act} \left(\bigwedge_{\alpha \in A} 0 \leq \sigma_{A,\alpha} \leq 1 \right) \wedge \sum_{\alpha \in A} \sigma_{A,\alpha} = 1$ `// scheduler choice`

3 $\quad\quad \wedge \bigwedge_{i=1}^{n} \bigwedge_{s \in S} \bigwedge_{\alpha \in Act(s)} \left(\bigvee_{j=0}^{m-1} \tau_{i,s,\alpha} = j \right)$ `// stuttering choice`

4 $\quad\quad \wedge \bigwedge_{i=1}^{n} \bigwedge_{(s,j) \in S \times [m]} \bigwedge_{\alpha \in Act(s)} \bigwedge_{(s',j') \in succ(s,\alpha)} \varphi^{go}_{i,(s,j),\alpha,(s',j')}$

5 $\quad\quad \wedge \bigwedge_{i=1}^{n} \bigwedge_{(s,j) \in S \times [m]} \bigwedge_{\alpha \in Act(s)} \bigwedge_{(s',j') \in succ(s,\alpha)} \varphi^{tr}_{i,(s,j),\alpha,(s',j')}$

6 $\quad \varphi_{sem} :=$ `Sem`$(\mathcal{M}, n, \varphi^{nq})$ `// semantics of` φ^{nq}

7 $\quad$ **foreach** $i = 1, \dots, l$ **do** `// encode state quantifiers`

8 $\quad\quad$ **if** $Q_i = \forall$ **then** $A_i :=$ "$\bigwedge_{s_i \in S}$" **else** $A_i :=$ "$\bigvee_{s_i \in S}$"

9 $\quad \varphi_{tru} := A_1 \dots A_l \left(h_{((s_{k_1},0),\dots,(s_{k_n},0)),\varphi^{nq}} \right)$ `// truth of input formula`

10 $\quad$ **if** $check(\varphi_{sch} \wedge \varphi_{sem} \wedge \varphi_{tru}) = SAT$ **then return** $TRUE$ **else return** $FALSE$

$Act(s) = Act(s')$, then $act(s, \alpha) = act(s', \alpha)$ for all $\alpha \in Act(s)$, and (2) stutter quantification ranges over m-bounded counting stutter-schedulers.

These restrictions were chosen to achieve decidability but still be expressive enough for our applications.

For simplicity, here we describe the case for a single scheduler quantifier, but the algorithm can be extended to an arbitrary number of scheduler quantifiers. Additionally, we only describe the algorithm for existential scheduler and stutter quantification. The extension to purely universal quantification is straightforward; we will discuss the handling of quantifier alternation in Sec. 6.

Our AHyperPCTL model checking method adapts the HyperPCTL algorithm [2] with two major extensions: (1) we consider probabilistic memoryless schedulers instead of deterministic memoryless ones and (2) we support stuttering. Assume in the following an MDP $\mathcal{M} = (S, Act, \mathbf{P}, \mathsf{AP}, L)$, a memory bound m for stutter-schedulers, and an input AHyperPCTL formula φ. Our method generates a quantifier-free real-arithmetic formula $\varphi_{sch} \wedge \varphi_{tru} \wedge \varphi_{sem}$ that is satisfiable if and only if $\mathcal{M} \models \varphi$ (under the above restrictions on the domains of schedulers and stutter-schedulers). The main method (Alg. 1) generates this encoding.

1) In φ_{sch} (Lines 2–5) we encode the scheduler probabilities and counting stutter-scheduler choices. We use real-valued variables $\sigma_{A,\alpha}$ to encode the probability of choosing α in state s with $Act(s) = A$, and variables $\tau_{i,s,\alpha}$ with domain $[m]$ (m being the stutter-scheduler memory bound) to represent the stuttering duration for state s and action α under the ith stutter-scheduler quantifier.

For $\boldsymbol{s} = ((s_1, j_1), \dots, (s_n, j_n)) \in (S \times [m])^n$ we define $Act(\boldsymbol{s}) = Act(s_1) \times \dots \times Act(s_n)$. The calculation of successor states for the encoding of the temporal operators depends on the chosen stutterings. To describe possible successors, we use two mechanisms: (i) For each $\boldsymbol{s} \in (S \times [m])^n$ and $\boldsymbol{\alpha} \in Act(\boldsymbol{s})$ we define $succ(\boldsymbol{s}, \boldsymbol{\alpha})$ to be the set of all $\boldsymbol{s'} = ((s'_1, j'_1), \dots, (s'_n, j'_n)) \in (S \times [m])^n$ which

Algorithm 2: SMT encoding for the meaning of the non-quantified formula

Input: $\mathcal{M} = (S, Act, \mathbf{P}, \mathsf{AP}, L)$: MDP; n: number of experiments;
φ: quantifier-free AHyperPCTL formula or expression.
Output: SMT encoding of the meaning of φ for $\mathcal{M}$.

1 **Function** $\texttt{Sem}(\mathcal{M}, n, \varphi)$
2 **if** φ *is* $\mathbb{P}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ **then**
3 $E := \texttt{Sem}(\mathcal{M}, \varphi_1, n) \wedge \texttt{Sem}(\mathcal{M}, \varphi_2, n)$
4 **foreach** $\boldsymbol{s} = ((s_1, j_1), \ldots, (s_n, j_n)) \in (S \times [m])^n$ **do**
5 $E := E \wedge (h_{\boldsymbol{s},\varphi_2} \Rightarrow pr_{\boldsymbol{s},\varphi}{=}1) \wedge \big((\neg h_{\boldsymbol{s},\varphi_1} \wedge \neg h_{\boldsymbol{s},\varphi_2}) \Rightarrow pr_{\boldsymbol{s},\varphi}{=}0\big)$
6 $E := E \wedge \Big[\Big[h_{\boldsymbol{s},\varphi_1} \wedge \neg h_{\boldsymbol{s},\varphi_2}\Big] \Rightarrow \Big[pr_{\boldsymbol{s},\varphi} = \sum_{\boldsymbol{\alpha} \in Act(\boldsymbol{s})} \sum_{\boldsymbol{s}' \in succ(\boldsymbol{s},\boldsymbol{\alpha})} \big(\prod_{i=1}^{n} \sigma_{Act(s_i),\alpha_i} \cdot go_{i,\boldsymbol{s}_i,\alpha_i,\boldsymbol{s}'_i} \cdot tr_{i,\boldsymbol{s}_i,\alpha_i,\boldsymbol{s}'_i}\big) \cdot pr_{\boldsymbol{s}',\varphi} \wedge \Big[pr_{\boldsymbol{s},\varphi}{>}0 \Rightarrow \bigvee_{\boldsymbol{\alpha} \in Act(\boldsymbol{s})} \bigvee_{\boldsymbol{s}' \in succ(\boldsymbol{s},\boldsymbol{\alpha})} \big(\prod_{i=1}^{n} \sigma_{Act(s_i),\alpha_i} \cdot go_{i,\boldsymbol{s}_i,\alpha_i,\boldsymbol{s}'_i}{>}0 \wedge (h_{\boldsymbol{s}',\varphi_2} \vee d_{\boldsymbol{s},\varphi_2}{>}d_{\boldsymbol{s}',\varphi_2})\big)\Big]\Big]\Big]$
7 **else if** ...
8 **return** E

under *some* stutter-scheduler could be successors of $\boldsymbol{s}$ under $\boldsymbol{\alpha}$, i.e., such that

$$\forall 1 \leq i \leq n.\ ((j_i < m-1 \wedge s_i = s'_i \wedge j'_i = j_i + 1) \vee (\mathbf{P}(s_i, \alpha_i, s'_i) > 0 \wedge j'_i = 0))\,.$$

(ii) For each $1 \leq i \leq n$, $(s, j) \in (S \times [m])$, $\alpha \in Act(s)$, and $(s', j') \in succ(s, \alpha)$ we define a pseudo-Boolean variable $go_{i,(s,j),\alpha,(s',j')}$ as well as a real variable $tr_{i,(s,j),\alpha,(s',j')}$ and define the formulas

$$\begin{aligned}
\varphi_{go_{i,(s,j),\alpha,(s',j')}} &= (go_{i,(s,j),\alpha,(s',j')}{=}0 \vee go_{i,(s,j),\alpha,(s',j')}{=}1) \wedge \big(go_{i,(s,j),\alpha,(s',j')}{=}1 \\
&\quad \leftrightarrow ((j < \tau_{i,s,\alpha} \wedge j' = j+1) \vee (j \geq \tau_{i,s,\alpha} \wedge j' = 0))\big) \\
\varphi_{tr_{i,(s,j),\alpha,(s',j')}} &= (j' = j+1 \wedge tr_{i,(s,j),\alpha,(s',j')} = 1) \vee \\
&\quad (j' = 0 \wedge tr_{i,(s,j),\alpha,(s',j')} = \mathbf{P}(s, \alpha, s'))\ .
\end{aligned}$$

2) In φ_{sem} (Line 6) we encode the semantics of the quantifier-free part φ^{nq} of the input formula by calling Alg. 2. The truth of each Boolean subformula φ' of φ^{nq} at state sequence $\boldsymbol{s}$ is encoded in a Boolean variable $h_{\boldsymbol{s},\varphi'}$. We also define variables $hInt_{\boldsymbol{s},\varphi'}$ for the integer encoding (i.e., 0 or 1) of $h_{\boldsymbol{s},\varphi'}$, and real-valued variables $pr_{\boldsymbol{s},\varphi''}$ for values of probability expressions φ''.

3) In φ_{tru} (Lines 7–9) we state the truth of the input formula by first encoding the state quantifiers (Lines 7–8) and then stating the truth of the quantifier-free part φ^{nq} under all necessary state quantifier instantiations $(s_1,\ldots,s_l)$, i.e., where experiment $i{\in}\{1,\ldots,n\}$ starts in state s_{k_i} and stutter-scheduler mode 0 (Line 9).

In Algorithm 2, we recursively encode the meaning of atomic propositions and Boolean, temporal, arithmetic and probabilistic operators. Due to space

constraints, here we present only the encoding for the temporal operator "until", and refer to the extended version [21] for the full algorithm.

For the encoding of the probability $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$ that $\varphi_1 \mathcal{U} \varphi_2$ is satisfied along the executions starting in state $\boldsymbol{s} \in (S \times [m])^n$, the interesting case, where φ_1 holds in $\boldsymbol{s}$ but φ_2 does not, is in Line 6. The probability is a sum over all possible actions and potential successor states. Each summand is a product over (i) the probability of choosing the given action tuple, (ii) pseudo-Boolean values which encode whether the potential successor state is indeed a successor state under the encoded stutter-schedulers, (iii) real variables encoding the probability of moving to the given successors under the encoded stutter-schedulers, and (iv) the probability to satisfy the until formula from the successor state. We use real variables $d_{\boldsymbol{s},\varphi,\boldsymbol{\tau}}$ to assure that finally a φ_2-state will be reached on all paths whose probabilities we accumulate.

Our SMT encoding is a Boolean combination of Boolean variables, non-linear real constraints, and linear integer constraints. Our linear integer constraints can be implemented as linear real constraints. The non-linear real constraints stem from the encoding of the probabilistic schedulers, not from the encoding of the stutter-schedulers. We check whether there exists an assignment of the variables such that the encoding is satisfied. SMT solving for non-linear real arithmetic without quantifier alternation has been proven to be solvable in exponential time in the number of variables [11,24]. However, all available tools run in doubly exponential time in the number of variables [4,16,26]. The number of variables of our encoding is exponential in the number of stutter quantifiers, and polynomial in the size of the formula, the number of states and actions of the model, and the memory size for the stutter-schedulers. Hence, in practice, our implementation is triple exponential in the size of the input. This yields an upper bound on the complexity of model checking the considered fragment.

The size of the created encoding is exponential in the number of stutter-schedulers and polynomial in the size of the AHyperPCTL formula, the number of states and actions of the model and the memory-size for the stutter-schedulers.

6 Implementation and Evaluation

We implemented the model checking algorithm described in Section 5 based on the existing implementation for HyperPCTL, using the SMT-solver Z3 [25]. We performed experiments on a PC with a 3.60GHz i7 processor and 32GB RAM. Our implementation and case studies are available at https://github.com/carolinager/A-HyperProb. It is important to note that checking the constructed SMT formula is more complicated than in the case for HyperPCTL, since the SMT formula contains non-linear real constraints due to the probabilistic schedulers, whereas the SMT formula for HyperPCTL contains only linear real arithmetic.

We optimized our implementation by reducing the number of variables as described in [20] based on the quantifiers relevant for the encoding of the considered subformula. However, for interesting properties like the properties presented in Section 4, where we want to compare probabilities in different executions, we

Case study		Running time (s)			SMT Solver			Model	
		Enc.	**Solving**	**Total**	result	#variables	#subform.	#states	#transitions
CE	$m = 2,\ h = (0, 1)$	0.25	15.82	16.07	sat	829	341	7	9
	$m = 2,\ h = (0, 2)$	0.38	DNF	-	-	1287	447	9	12
TL	$m = 2,\ k = 1$	0.99	DNF	-	-	3265	821	15	23
ACDB	$m = 2$	9.17	OOM	-	-	14716	1284	24	36

Table 1: Experimental results. **CE**: Classic example, **TL**: Timing leakage, **ACDB**: Output information leak. DNF: did not finish, OOM: out of memory.

nevertheless have to create a variable encoding the validity of a subformula at a state for $(|S| \cdot m)^n$ combinations of states for each subformula.

All example applications presented in Sec. 1 and 4 consist of universal scheduler and state quantification, and existential stutter quantification. However, our implementation is restricted to existential scheduler and stutter quantification. Allowing arbitrary quantifiers for scheduler and stutter quantification is also possible in theory, but we found it to be infeasible in practice. For universal quantifiers we would need to check whether the SMT encoding holds for all possible assignments of the variables encoding the schedulers and stutter-schedulers, while for all other variables we only check for existence. This would yield an SMT instance with quantifier alternation, which is considerably more difficult than the current encoding with purely existential quantification [14]. Alternatively, we can encode the semantics for each possible combination of stutter-schedulers separately, meaning that we have to use variables $h_{\boldsymbol{s},\varphi,\boldsymbol{\tau}}$ encoding the truth of φ at $\boldsymbol{s}$ under $\boldsymbol{\tau}$. This makes the number of variables exponential in the number of stutter quantifiers and the number of states and actions of the model, and polynomial in the size of the formula and the memory-size for stuttering. As a result, the implementation scales very badly, even after decreasing the number of variables via an optimization based on the relevant quantifiers. For all our applications, creating the semantic encoding exceeded memory after 30 minutes at the latest in this case.

Since neither option is a viable solution, for our case studies we instead consider the presented formulas with existential instead of universal scheduler quantification. In future work, it would be worth to explore how one could employ quantifier elimination to generate a set of possible schedulers from one scheduler instance satisfying the existential quantification.

The results of our case studies are presented in Table 1. Our first case study, **CE**, is the classic example presented in Sec. 1. We compare executions with different initial values h_1 and h_2, denoted by $h = (h_1, h_2)$. For $h = (0, 1)$, the property can already be satisfied for the smallest non-trivial memory size $m = 2$. For higher values of h_2, however, the SMT solver does not finish solving after 1 hour. The second case study, **TL**, is the side-channel timing leak described in Sec. 4. We found that already for encryption key length $k = 1$, and memory size $m = 2$, the SMT solver did not finish after 1 hour even for a smaller formula, where we restrict the conjunction to the case $l = 0$. Our third case study, **ACDB**,

is the output information leak presented in Sec. 4. Here, the SMT solving exceeds memory after 18 minutes, even if we check only part of the conjunction.

We see several possibilities to improve the scalability of our implementation. Firstly, we could experiment with different SMT solvers, like cvc5 [6]. Secondly, we could parallelize the construction of encodings for different stutter-schedulers, and possibly re-use sub-encodings that are the same for multiple stutter-schedulers. Another possibility would be to turn towards less accurate methods, and employ Monte Carlo or statistical model checking approaches.

7 Conclusions

We proposed a new logic, called AHyperPCTL, which is, to our knowledge, the first asynchronous probabilistic hyperlogic. AHyperPCTL extends HyperPCTL by quantification over stutter-schedulers, which allow to specify when the execution of a program should stutter. This allows to verify whether there exist stuttering variants of programs that would prevent information leaks, by comparing executions of different stuttering variations of a program. AHyperPCTL subsumes HyperPCTL. Therefore, the AHyperPCTL model checking problem on MDPs is, in general, undecidable. However, we showed that the model checking is decidable if we restrict the quantification to probabilistic memoryless schedulers and deterministic stutter-schedulers with bounded memory. Since our prototype implementation does not scale well, future work could investigate the use of other SMT solvers, statistical model checking, or Monte Carlo methods, as well as a feasible extension to quantifier alternation for scheduler and stutter quantifiers.

Acknowledgements Lina Gerlach is supported by the DFG RTG 2236/2 *UnRAVeL*. Ezio Bartocci is supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19018]. This work is also partially sponsored by the United States NSF SaTC awards 2245114 and 2100989.

References

1. Ábrahám, E., Bartocci, E., Bonakdarpour, B., Dobe, O.: Parameter synthesis for probabilistic hyperproperties. In: Proc. of LPAR-23. 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. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. J. Log. Algebraic Methods Program. **119**,

100633 (2021). https://doi.org/10.1016/j.jlamp.2020.100633, https://doi.org/10.1016/j.jlamp.2020.100633
5. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
6. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Proc. of TACAS 2022: the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
7. Bartocci, E., Ferrère, T., Henzinger, T.A., Nickovic, D., da Costa, A.O.: Flavors of sequential information flow. In: Finkbeiner, B., Wies, T. (eds.) Proc. of VMCAI 2022: the 23rd International Conference on Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 13182, pp. 1–19. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_1
8. Baumeister, J., Coenen, N., Bonakdarpour, B., Sánchez, B.F.C.: A temporal logic for asynchronous hyperproperties. In: Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV). pp. 694–717 (2021)
9. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: Silva, A., Leino, K.R.M. (eds.) Proc. of CAV 2021: the 33rd International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 12759, pp. 694–717. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_33
10. Beutner, R., Finkbeiner, B.: A logic for hyperproperties in multi-agent systems. CoRR **abs/2203.07283** (2022). https://doi.org/10.48550/arXiv.2203.07283, https://doi.org/10.48550/arXiv.2203.07283
11. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 739–764. IOS Press (2021). https://doi.org/10.3233/FAIA201002, https://doi.org/10.3233/FAIA201002
12. Bozzelli, L., Peron, A., Sánchez, C.: Asynchronous extensions of hyperLTL. In: Proc. of LICS 2021: the 36th Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 1–13. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470583
13. Bozzelli, L., Peron, A., Sánchez, C.: Expressiveness and decidability of temporal logics for asynchronous hyperproperties. In: Klin, B., Lasota, S., Muscholl, A. (eds.) Proc. of CONCUR 2022: the 33rd International Conference on Concurrency Theory. LIPIcs, vol. 243, pp. 27:1–27:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.27
14. Brown, C.W., Davenport, J.H.: The complexity of quantifier elimination and cylindrical algebraic decomposition. In: Wang, D. (ed.) Symbolic and Algebraic Computation, International Symposium, ISSAC 2007, Waterloo, Ontario, Canada, July 28 - August 1, 2007, Proceedings. pp. 54–60. ACM (2007). https://doi.org/10.1145/1277548.1277557, https://doi.org/10.1145/1277548.1277557
15. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Proc. of the 3rd Conf. on Principles of Security and Trust POST. pp. 265–284 (2014)
16. Collins, G.E.: Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Barkhage, H. (ed.) Automata

Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, May 20-23, 1975. Lecture Notes in Computer Science, vol. 33, pp. 134–183. Springer (1975). https://doi.org/10.1007/3-540-07407-4_17, https://doi.org/10.1007/3-540-07407-4_17

17. 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
18. 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
19. Dobe, O., Wilke, L., Ábrahám, E., Bartocci, E., Bonakdarpour, B.: Probabilistic hyperproperties with rewards. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) Proc. of NFM 2022: the 14th International Symposium on NASA Formal Methods. LNCS, vol. 13260, pp. 656–673. Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_35
20. Dobe, O., Ábrahám, E., Bartocci, E., Bonakdarpour, B.: Model checking hyperproperties for markov decision processes. Information and Computation **289**, 104978 (2022). https://doi.org/10.1016/j.ic.2022.104978, special Issue on 11th Int. Symp. on Games, Automata, Logics and Formal Verification
21. Gerlach, L., Dobe, O., Ábrahám, E., Bartocci, E., Bonakdarpour, B.: Introducing asynchronicity to probabilistic hyperproperties. CoRR **abs/2307.05282** (2023)
22. Guernic, G.L.: Automaton-based confidentiality monitoring of concurrent programs. In: Proc. of CSF 2007: the 20th IEEE Computer Security Foundations Symposium. pp. 218–232. IEEE Computer Society (2007). https://doi.org/10.1109/CSF.2007.10
23. Hsu, T., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: Bounded model checking for asynchronous hyperproperties. CoRR **abs/2301.07208** (2023). https://doi.org/10.48550/arXiv.2301.07208
24. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series, Springer (2016). https://doi.org/10.1007/978-3-662-50497-0, https://doi.org/10.1007/978-3-662-50497-0
25. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Proc. of TACAS 2008: the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. de Moura, L.M., Jovanovic, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7737, pp. 1–12. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_1, https://doi.org/10.1007/978-3-642-35873-9_1
27. Ngo, T.M., Stoelinga, M., Huisman, M.: Confidentiality for probabilistic multithreaded programs and its verification. In: Jürjens, J., Livshits, B., Scandariato, R. (eds.) Proc. of ESSoS 2013: the 5th International Symposium on Engineering Secure Software and Systems. LNCS, vol. 7781, pp. 107–122. Springer (2013). https://doi.org/10.1007/978-3-642-36563-8_8

28. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009). https://doi.org/10.3233/JCS-2009-0352
29. Smith, G.: Probabilistic noninterference through weak probabilistic bisimulation. In: Proc. of (CSFW-16 2003: the 16th IEEE Computer Security Foundations Workshop. pp. 3–13. IEEE Computer Society (2003). https://doi.org/10.1109/CSFW.2003.1212701
30. Wang, Y., Nalluri, S., Bonakdarpour, B., Pajic, M.: Statistical model checking for hyperproperties. In: Proc. of CSF 2021: the 34th IEEE Computer Security Foundations Symposium. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00009
31. Wang, Y., Zarei, M., Bonakdarpour, B., Pajic, M.: Statistical verification of hyperproperties for cyber-physical systems. ACM Trans. Embed. Comput. Syst. **18**(5s), 92:1–92:23 (2019). https://doi.org/10.1145/3358232