A Temporal Logic for Asynchronous Hyperproperties

Jan Baumeister[1], Norine Coenen[1], Borzoo Bonakdarpour[2], Bernd Finkbeiner[1], and César Sánchez[3](✉)

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
[2] Michigan State University, East Lansing, USA
[3] IMDEA Software Institute, Madrid, Spain
cesar.sanchez@imdea.org

Abstract. *Hyperproperties* are properties of computational systems that require more than one trace to evaluate, e.g., many information-flow security and concurrency requirements. Where a trace property defines a set of traces, a hyperproperty defines a set of sets of traces. The temporal logics HyperLTL and HyperCTL* have been proposed to express hyperproperties. However, their semantics are *synchronous* in the sense that all traces proceed at the same speed and are evaluated at the same position. This precludes the use of these logics to analyze systems whose traces can proceed at different speeds and allow that different traces take stuttering steps independently. To solve this problem in this paper, we propose an *asynchronous* variant of HyperLTL. On the negative side, we show that the model-checking problem for this variant is undecidable. On the positive side, we identify a decidable fragment which covers a rich set of formulas with practical applications. We also propose two model-checking algorithms that reduce our problem to the HyperLTL model-checking problem in the synchronous semantics.

1 Introduction

Hyperproperties [8] extend the conventional notion of trace properties [1] from a set of traces to a set of sets of traces. In other words, a hyperproperty stipulates a system property and not the property of just individual traces. Many interesting requirements in computing systems are hyperproperties and cannot be expressed by trace properties. Examples include (1) a wide range of information-flow security policies such as *noninterference* [14] and *observational determinism* [28],

This work was funded in part by Madrid Regional Government under project "S2018/TCS-4339 (BLOQUES-CM)", by Spanish National Project "BOSCO (PGC2018-102210-B-100)", by the German Research Foundation (DFG) as part of the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), by the European Research Council (ERC) Grant OSARES (No. 683300), and by the United Stated NSF SaTC Award 2100989.

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 694–717, 2021.
https://doi.org/10.1007/978-3-030-81685-8_33

```
ℓ1  int l = 0;
ℓ2
ℓ3  if (h = 0)
ℓ4      l := l + 1;
ℓ5  else
ℓ6      l := 1;
```

Fig. 1. Program P_1

```
ℓ1  int l = 0;
ℓ2
ℓ3  if (h = 0)
ℓ4      reg:= l + 1;
ℓ5      l := reg;
ℓ6  else
ℓ7      l := 1;
```

Fig. 2. Program P_2

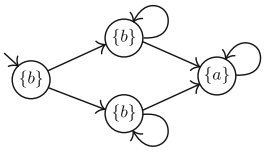
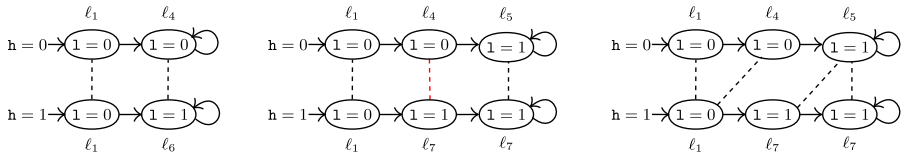


Fig. 3. $\mathcal{K}$ with a self-loop

(2) sensitivity and robustness requirements in cyber-physical systems [27], and (3) consistency conditions such as *linearizability* in concurrent data structures [5].

HyperLTL [7] is a temporal logic for hyperproperties that enriches LTL with quantifiers allowing explicit and simultaneous quantification over multiple execution traces. For example, the observational determinism security policy [28] stipulates that any two executions that start in two *low-equivalent* states (i.e., states whose value of publicly observable variables are the same), should remain in low-equivalent states. This property can be expressed in HyperLTL as the following formula, called φ_{OD},$\forall\pi.\forall\pi'.(l_\pi \leftrightarrow l_{\pi'}) \rightarrow \square(l_\pi \leftrightarrow l_{\pi'})$. However, the semantics of HyperLTL (and other formal languages for hyperproperties) is *synchronous*, meaning that they completely abstract away the notion of time passage. In HyperLTL, all traces proceed at the same speed, as all temporal operators move the position on all traces simultaneously. Consider the program P_1 in Fig. 1, where input values `0` and `1` are possible for *high-secret* variable `h`. This renders two possible traces shown in Fig. 4a that satisfy φ_{OD}.

The synchronous semantics of HyperLTL has a shortcoming which has practical implications as well: formulas are not invariant under *stuttering*. Note that, contrary to LTL, disallowing the use of $\bigcirc$ does not make the formula invariant under stuttering, as traces can still stutter independently. This limits the scope of application of HyperLTL to only those settings where different traces can be perfectly aligned. For example, consider program P_2 in Fig. 2, where line ℓ_4 in P_1 is refined to its intermediate code using a register that stores the value `l` $+ 1$ and then stores this value in memory location `l` in lines ℓ_4 and ℓ_5, respectively. Applying the synchronous semantics of HyperLTL results in declaring a violation of φ_{OD} in the second position. This, however, is not an accurate interpretation of φ_{OD} (assuming that an attacker only has access to the memory footprint and not the CPU registers or a timing channel), as the two traces are stutter equivalent with respect to the state of variable `l`. In fact, the synchronous semantics of HyperLTL may incorrectly identify good programs as bad because it ignores the notion of relative time between traces. This problem is generally amplified in Kripke structures where self-loops correspond to non-deterministic choices that model that the system may remain in a state for some arbitrary time. For instance, consider $\mathcal{K}$ in Fig. 3 and HyperLTL formula $\forall\pi.\forall\pi'.((b_\pi \leftrightarrow b_{\pi'})\ \mathcal{U}\ \square(a_\pi \leftrightarrow a_{\pi'}))$. Only pairs of traces that take the self-loop the same number of times satisfy this formula. However, since the goal of employing a self-loop is typically to make the duration of staying in a state irrelevant, this semantics is too restrictive.



(a) P_1 satisfies φ_{OD} under synchronous sems.

(b) P_2 violates φ_{OD} under synchronous semantics

(c) P_2 satisfies φ_{OD} under asynchronous semantics.

Fig. 4. Synchronous vs. asynchronous semantics for HyperLTL.

Besides HyperLTL, other logics have been proposed that allow trace quantification, for example, H_μ [15], which extends the linear time μ-calculus [3] with path quantifiers and indexed next operators. For H_μ, the model-checking problem is in general undecidable, but two fragments, the k-synchronous, k-context bounded fragments, have been identified for which model checking remains decidable [15].

In this paper, we propose an asynchronous temporal logic for hyperproperties. Our main motivation is to be able to reason about execution traces according to the relative order of the sequences of actions in each trace but not about the duration of each action. Software is inherently asynchronous, and so is hardware in many cases if one abstracts the execution platform or many features of the execution platform like pipelines, caches, memory contention, etc. We call our temporal logic *Asynchronous HyperLTL* or in short, A-HLTL. The key addition is the notion of *trajectory* that controls the relative speed at which traces progress by chosing at each instant which traces move and which traces stutter. For example, the trajectory shown in Fig. 4c for the two traces of the program in Fig. 2 allows the lower trace to stutter in the first position while the upper trace advances. On the contrary, in the third position, the upper trace stutters while the lower trace moves from the second to the third position. This trajectory enables identification of stutter equivalence of the two traces with respect to state variable `l` and, hence, successful verification of observational determinism. In order to reflect the notion of trajectories in our logic, we lift the syntax of HyperLTL by allowing a trajectory modality. This way, the corresponding formula for observational determinism in A-HLTL is the following:

$$\varphi_{\mathsf{OD}} \stackrel{\text{def}}{=} \forall\pi.\forall\pi'.\mathsf{E}.(li_\pi \leftrightarrow li_{\pi'}) \rightarrow \Box(lo_\pi \leftrightarrow lo_{\pi'})$$

where E denotes the *existence* of a trajectory for temporal operator $\Box$. The A-HLTL formula for the Kripke structure in Fig. 3 is $\forall\pi.\forall\pi'.\mathsf{E}.((b_\pi \leftrightarrow b_{\pi'})\ \mathcal{U}\ \Box(a_\pi \leftrightarrow a_{\pi'}))$. A-HLTL allows us to reason about relational properties between two different systems that differ on timing, like for example, translation validation [22], which relates executions of the target code with the source code with respect to a (trace or hyper) property.

We show an encoding of the PCP problem into model-checking a formula of the shape $\forall\pi.\forall\pi'.\mathsf{E}.(\Box\psi_1(\pi,\pi') \wedge \Diamond\psi_2(\pi,\pi'))$, which implies that model-checking A-HLTL is undecidable, even for the universal fragment. On the positive side,

we show two decidable fragments of A-HLTL. The first algorithm is based on a *stuttering construction* in which we modify the Kripke structure to accept all stuttering expansions of the original paths. This algorithm can handle fragment $\forall \pi_1 \dots \pi_n.\mathsf{E}.\psi$, where the ψ is a *phase formula*, a class of safety formulas that appear in many hyperproperties and are the building block of expressing trace equivalence. Our second algorithm uses an *acceleration construction* to convert a finite sequence of transitions that do not change phase, into a single transition. This algorithm is able to handle formulas with arbitrary quantification but a simpler kind of phase formulas. A-HLTL is, thus, the first logic for hyperproperties that can express the major asynchronous hyperproperties of interest within decidable fragments. Moreover, A-HLTL is the first logic for asynchronous hyperproperties with a practical model checking algorithm. Both algorithms use internally HyperLTL model-checking as a building block. However, the reduction from A-HLTL model-checking into HyperLTL requires modifying both the formula and the model in a highly non-trivial way, to encode the exitence of trajectories. The choice of using HyperLTL model-checking as a building block is based on the existence of tools, but it does not imply that asynchronous properties of interest can be expressed in HyperLTL directly.

We have evaluated the stuttering construction on two sets of cases studies: a range of compiler optimizations and an SPI bus protocol. In both case studies, we were able to prove system correctness using our reduction from A-HLTL to synchronous HyperLTL.

Organization. The rest of the paper is structured as follows. Section 2 contains the preliminaries, and Sect. 3 introduces A-HLTL and presents examples of properties expressible in A-HLTL. Section 4 describes the decidable fragments and present procedures for the model-checking problem. Section 5 shows that the model-checking problem for general A-HLTL formulas is undecidable and present the lower-bound complexity. Experimental results are presented in Sect. 6. Finally, Sect. 7 discusses the related work, while Sect. 8 concludes. Detailed proofs appear in the longer version of this paper in [4].

2 Preliminaries

Let AP be a set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the *alphabet*, where we call each element of Σ a *letter*. A *trace* is an infinite sequence $\sigma = a_0 a_1 \cdots$ of letters from Σ. We denote the set of all infinite traces by Σ^ω. We use $\sigma(i)$ for a_i and σ^i for the suffix $a_i a_{i+1} \cdots$. A *pointed trace* is a pair (σ, p), where $p \in \mathbb{N}_0$ is a natural number (called the *pointer*). Pointed traces allow to traverse a trace by moving the pointer. Given a pointed trace (σ, p) and $n > 0$, we use $(\sigma, p) + n$ as a short for $(\sigma, p + n)$. We denote the set of all pointed traces by $\mathsf{PTR} = \{(\sigma, p) \mid \sigma \in \Sigma^\omega \text{ and } p \in \mathbb{N}_0\}$.

Two pointed traces (σ, p) and (σ', p') are *stuttering equivalent* if there are two infinite sequences of indices $p = i_0 < i_1 \dots$ and $p' = j_0 < j_1 \dots$ such that for all $k \geq 0$ and for all $l \in [i_k, i_{k+1})$ and $l' \in [j_k, j_{k+1})$, $\sigma(l) = \sigma'(l')$. A pointed trace

(σ', p') is a *stuttering expansion* of (σ, p) if there is a sequence $p' = j_0 < j_1 < \ldots$ such that for all $k \geq 0$ and for all $l \in [j_k, j_{k+1})$, $\sigma(p+k) = \sigma'(l)$. We say that σ is stuttering equivalent to σ' if $(\sigma, 0)$ is stuttering equivalent to $(\sigma', 0)$, and that σ' is a stuttering expansion of σ if $(\sigma', 0)$ is a stuttering expansion of $(\sigma, 0)$.

A *Kripke structure* is a tuple $\mathcal{K} = \langle S, S_{init}, \delta, L \rangle$, where S is a set of states, $S_{init} \subseteq S$ is the set of initial states, $\delta \subseteq S \times S$ is a transition relation, and $L : S \to \Sigma$ is a labeling function on the states of $\mathcal{K}$. We require that for each $s \in S$, there exists $s' \in S$, such that $(s, s') \in \delta$.

A *path* of a Kripke structure is an infinite sequence of states $s(0)s(1)\cdots \in S^\omega$, such that $s(0) \in S_{init}$ and $(s(i), s(i+1)) \in \delta$, for all $i \geq 0$. A *trace* of a Kripke structure is a trace $\sigma(0)\sigma(1)\sigma(2)\cdots \in \Sigma^\omega$, such that there exists a path $s(0)s(1)\cdots \in S^\omega$ with $\sigma(i) = L(s(i))$ for all $i \geq 0$. Abusing notation we use $\sigma = L(\rho)$ to denote that σ is the trace corresponding to path ρ. We denote by $\mathsf{Traces}(\mathcal{K}, s)$ the set of all traces of $\mathcal{K}$ with paths that start in state $s \in S$, We denote by $\mathsf{Traces}(\mathcal{K}, A)$ the set of all traces that start from some state in $A \subseteq S$ and $\mathsf{Traces}(\mathcal{K})$ as a short for $\mathsf{Traces}(\mathcal{K}, S_{init})$.

HyperLTL. HyperLTL [7] is a temporal logic that extends LTL [19,21] for hyperproperties, which allows reasoning about multiple execution traces simultaneously. The syntax of HyperLTL is:

$$\begin{aligned} \varphi &::= \exists \pi.\varphi \mid \forall \pi.\varphi \mid \psi \\ \psi &::= a_\pi \mid \psi \vee \psi \mid \neg\psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi \end{aligned}$$

where π is a *trace variable* from an infinite supply of trace variables. The intended meaning of a_π is that proposition $a \in \Sigma$ holds in the current time in trace π. Trace quantifiers $\exists\pi$ and $\forall\pi$ allow reasoning simultaneously about different traces of the computation. Atomic predicates a_π refer to a single trace π. Given a HyperLTL formula φ, we use $\mathsf{Vars}(\varphi)$ for the set of trace variables quantified in φ. A formula φ is well-formed if for all atoms a_π in φ, π is quantified in φ (i.e., $\pi \in \mathsf{Vars}(\varphi)$) and if no trace variable is quantified twice in φ. Given a set of traces T, the semantics of a HyperLTL formula φ is defined in terms of trace assignments, which is a (partial) map from trace variables to indexed traces $\Pi : \mathsf{Vars}(\varphi) \rightharpoonup \mathsf{PTR}$. The trace assignment with empty domain is denoted by $\Pi_\emptyset$. We use $Dom(\Pi)$ for the subset of $\mathsf{Vars}(\varphi)$ for which Π is defined. Given a trace assignment Π, a trace variable π, a trace σ and a pointer p, we denote by $\Pi[\pi \mapsto (\sigma, p)]$ the assignment that coincides with Π for every trace variable except for π, which is mapped to (σ, p). Also, we use $\Pi + n$ to denote the trace assignment Π' such that $\Pi'(\pi) = \Pi(\pi) + n$ for all $\pi \in Dom(\Pi) = Dom(\Pi')$. The semantics of HyperLTL is:

$$\begin{aligned} \Pi \models_T \exists\pi.\varphi \quad &\text{iff} \quad \text{for some } \sigma \in T,\ \Pi[\pi \mapsto (\sigma, 0)] \models_T \varphi \\ \Pi \models_T \forall\pi.\varphi \quad &\text{iff} \quad \text{for all } \sigma \in T,\ \Pi[\pi \mapsto (\sigma, 0)] \models_T \varphi \\ \Pi \models_T \psi \quad &\text{iff} \quad \Pi \models \psi \\ \Pi \models a_\pi \quad &\text{iff} \quad a \in \sigma(p),\ \text{where } (\sigma, p) = \Pi(\pi) \end{aligned}$$

$$
\begin{array}{lll}
\Pi \models_T \psi_1 \vee \psi_2 & \text{iff} & \Pi \models_T \psi_1 \text{ or } \Pi \models_T \psi_2 \\
\Pi \models \neg\psi & \text{iff} & \Pi \not\models \psi \\
\Pi \models \bigcirc\psi & \text{iff} & (\Pi + 1) \models \psi \\
\Pi \models \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \text{for some } j \geq 0 \;\; (\Pi + j) \models \psi_2 \\
 & & \quad \text{and for all } 0 \leq i < j, (\Pi + i) \models \psi_1
\end{array}
$$

Note that quantifiers assign traces to trace variables and set the pointer to the initial position 0. We say that a set of traces T is a model of a HyperLTL formula φ, denoted $T \models \varphi$ whenever $\Pi_\emptyset \models_T \varphi$. A Kripke structure $\mathcal{K}$ is a model of a HyperLTL formula φ, denoted by $\mathcal{K} \models \varphi$, whenever $\mathsf{Traces}(\mathcal{K}) \models \varphi$.

3 Asynchronous HyperLTL

We introduce a temporal logic A-HLTL as an extension of HyperLTL to express asynchronous hyperproperties.

Trajectories. To model the asynchronous passage of time, we now introduce the notion of a trajectory, which chooses when traces move and when they stutter. Let $\mathcal{V}$ be a set of trace variables and let $I \subseteq \mathcal{V}$. The I-successor of a trace assignment Π, denoted by $\Pi + I$, is the trace assignment Π' such that $\Pi'(\pi) = \Pi(\pi) + 1$ if $\pi \in I$ and $\Pi'(\pi) = \Pi(\pi)$ otherwise. That is, the pointers of indices in I advance by one step, while the others remain the same. A *trajectory* $t : t(0)t(1)t(2)\cdots$ for a formula φ is an infinite sequence of non-empty subsets of $\mathsf{Vars}(\varphi)$. Essentially, in each step of the trajectory one or more of the traces make progress. A trajectory is fair for a trace variable $\pi \in \mathsf{Vars}(\varphi)$ if there are infinitely many positions j such that $\pi \in t(j)$. A trajectory is fair if it is fair for all trace variables in $\mathsf{Vars}(\varphi)$. Given a trajectory t, by t^i, we mean the suffix $t(i)t(i+1)\cdots$. Furthermore, for a set of trace variables $\mathcal{V}$, we use $\mathsf{TRJ}_\mathcal{V}$ for set of all trajectories for indices from $\mathcal{V}$.

3.1 Syntax and Semantics of Asynchronous HyperLTL

The syntax of Asynchornous HyperLTL is:

$$
\begin{aligned}
\varphi &::= \exists\pi.\varphi \mid \forall\pi.\varphi \mid \mathsf{E}\psi \mid \mathsf{A}\psi \\
\psi &::= a_\pi \quad \mid \neg\psi \quad \mid \psi_1 \vee \psi_2 \mid \psi_1 \;\mathcal{U}\; \psi_2 \mid \bigcirc\psi
\end{aligned}
$$

where $a \in \mathsf{AP}$, π is a trace variable from an infinite supply $\mathcal{V}$ of trace variables, E is the existential trajectory modality and A is the universal trajectory modality. The intended meaning of E is that there is a trajectory that gives an interpretation of the relative passage of time between the traces for which the temporal formula that relates the traces is satisfied. Dualy, A means that for all trajectories, the resulting alignment makes the inner formula true. It is important

to note that there is no nesting of trajectory modalities and that all temporal operators in a formula are interpreted with respect to a single modality.

We use the usual syntactic sugar for Boolean operators $\mathit{true} \stackrel{\text{def}}{=} a_\pi \vee \neg a_\pi$, $\mathit{false} \stackrel{\text{def}}{=} \neg \mathit{true}$, $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$, and the syntactic sugar for temporal operators $\Diamond\varphi \stackrel{\text{def}}{=} \mathit{true}\,\mathcal{U}\,\varphi$, $\varphi_1 \rightarrow \varphi_2 \stackrel{\text{def}}{=} \neg\varphi_1 \vee \varphi_2$, and $\Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$, etc.

As before, we use trace assignments for the semantics of A-HLTL. Given (Π, t) where Π is a trace assignment and t a trajectory, we use $(\Pi, t) + 1$ for the successor of (Π, t) defined as (Π', t') where $t' = t^1$, and $\Pi'(\pi) = \Pi(\pi) + 1$ if $\pi \in t(0)$ and $\Pi'(\pi) = \Pi(\pi)$ otherwise. We use $(\Pi, t) + k$ as the k-th successor of (Π, t).

The satisfaction of an asynchronous HyperLTL formula φ over a trace assignment Π and a set of traces T, denoted by $\Pi \models_T \varphi$ is defined as follows:

$$
\begin{array}{rcll}
\Pi \models_T \exists \pi.\varphi & \text{iff} & \text{for some } \sigma \in T : \Pi[\pi \mapsto (\sigma, 0)] \models_T \varphi \\
\Pi \models_T \forall \pi.\varphi & \text{iff} & \text{for all } \sigma \in T : \Pi[\pi \mapsto (\sigma, 0)] \models_T \varphi \\
\Pi \models_T \mathsf{E}\psi & \text{iff} & \text{for some } t \in \mathsf{TRJ}_{Dom(\Pi)}.\, (\Pi, t) \models \psi \\
\Pi \models_T \mathsf{A}\psi & \text{iff} & \text{for all } t \in \mathsf{TRJ}_{Dom(\Pi)}.\, (\Pi, t) \models \psi \\
(\Pi, t) \models a_\pi & \text{iff} & a \in \Pi(\pi)(0) \\
(\Pi, t) \models \neg\psi & \text{iff} & (\Pi, t) \not\models \psi \\
(\Pi, t) \models \psi_1 \vee \psi_2 & \text{iff} & (\Pi, t) \models \psi_1 \text{ or } (\Pi, t) \models \psi_2 \\
(\Pi, t) \models \bigcirc\psi & \text{iff} & (\Pi, t) + 1 \models \psi \\
(\Pi, t) \models \psi_1\,\mathcal{U}\,\psi_2 & \text{iff} & \text{for some } i \geq 0 : (\Pi, t) + i \models \psi_2 \text{ and} \\
 & & \qquad \text{for all } j < i : (\Pi, t) + j \models \psi_1
\end{array}
$$

We say that a set T of traces satisfies a closed sentence φ, denoted by $T \models \varphi$, if $\Pi_\emptyset \models_T \varphi$. We say that a Kripke structure $\mathcal{K}$ satisfies an A-HLTL formula φ (and write $\mathcal{K} \models \varphi$) if and only if we have $\mathsf{Traces}(\mathcal{K}, S_{init}) \models \varphi$.

3.2 Examples of A-HLTL

We illustrate the expressive power of A-HLTL by introducing the asynchronous version of well-known properties.

Linearizability. [16] requires that any history of execution of a concurrent data structure (i.e., sequence of invocation and response by different threads) matches some sequential order of invocations and responses:

$$\varphi_{\mathsf{LNZ}} \stackrel{\text{def}}{=} \forall\pi.\exists\pi'.\mathsf{E}.\Box(\mathsf{history}_\pi \leftrightarrow \mathsf{history}_{\pi'})$$

where history denotes method invocations (and not the actual execution of the internal instructions of the concurrent library) by the different threads and the response observed, trace π ranges over the concurrent data structure and π' ranges over its sequential counterpart.

Goguen and Meseguer's Noninterference (GMNI). [14] stipulates that, for all traces, the low-observable output must not change when all high inputs are removed:

$$\varphi_{\mathsf{GMNI}} \stackrel{\text{def}}{=} \forall\pi.\exists\pi'.\mathsf{E}.(\Box\lambda_{\pi'}) \wedge \Box(lo_{\pi} \leftrightarrow lo_{\pi'})$$

where $\lambda_{\pi'}$ expresses that all of the high inputs in the current state of π' have dummy value λ, and denotes low-observable output proposition.

Not never Terminates. [18] requires that for every initial state, there is a terminating trace and a non-terminating trace:

$$\varphi_{\mathsf{NNT}} \stackrel{\text{def}}{=} \forall\pi.\exists\pi'.\exists\pi''.\mathsf{E}.(\pi[0] = \pi'[0] = \pi''[0]) \rightarrow (\Diamond\,\mathsf{term}_{\pi'} \wedge \Box\neg\mathsf{term}_{\pi''})$$

Termination-Insensitive Noninterference. [25] requires that for two executions that start from a low-observable states, information leaks are permitted if they are transmitted purely by the program's termination behavior. That is, the program may diverge on some high inputs and terminate on others:

$$\varphi_{\mathsf{TIN}} \stackrel{\text{def}}{=} \forall\pi.\forall\pi'.\mathsf{E}.\Big(l_{\pi} \leftrightarrow l_{\pi'}\Big) \rightarrow \left(\begin{array}{l}(\Box\neg\mathsf{term}_{\pi} \vee \Box\neg\mathsf{term}_{\pi'}) \vee \\ \Diamond(\mathsf{term}_{\pi} \wedge \mathsf{term}_{\pi'} \wedge l_{\pi} \leftrightarrow l_{\pi'})\end{array}\right)$$

Termination-Sensitive Noninterference. [2] Termination-sensitive noninterference is the same as termination insensitive, except that it forbids one trace to diverge and the other to terminate:

$$\varphi_{\mathsf{TSN}} \stackrel{\text{def}}{=} \forall\pi.\forall\pi'.\mathsf{E}.\Big(l_{\pi} \leftrightarrow l_{\pi'}\Big) \rightarrow \left(\begin{array}{l}(\Box\neg\mathsf{term}_{\pi} \wedge \Box\neg\mathsf{term}_{\pi'}) \vee \\ \Diamond(\mathsf{term}_{\pi} \wedge \mathsf{term}_{\pi'} \wedge l_{\pi} \leftrightarrow l_{\pi'})\end{array}\right)$$

4 Model-Checking A-HLTL

In this section, we show the decidability of the model-checking problem for two classes of A-HLTL formulas using two different algorithms:

(1) a *stuttering* construction in which we modify the Kripke structure $\mathcal{K}$ to accept all stuttering expansions of paths in $\mathcal{K}$; and
(2) an *acceleration* construction in which the modified Kripke structure accelerates jumping directly to the synchronization points.

In both cases the problem is reduced to model-checking HyperLTL formulas, which is known to be decidable [7,12]. We describe each construction separately.

4.1 The Stuttering Construction

We consider first A-HLTL formulas of the form $\forall\pi_1 \dots \pi_n.\mathsf{E}.\psi$. We will then extend our results to the $\exists^*$ fragment, to handle the A trajectory modality and to a larger collection of predicates. The class of temporal formulas ψ that we handle are called *admissible* formulas, and are defined as the Boolean combination of:

1. any number of state formulas, which may relate propositions p_{π_i} of different traces arbitrarily;
2. any number temporal formulas (called *monadic temporal formulas*), each of which only uses one trace variable and is invariant under stuttering (guaranteed for example by forbidding the use of $\bigcirc$), and
3. one *phase formula*, which is an invariant that can relate different traces in a restricted way (see below).

Given an admissible formula ψ, we use ψ_{ph} for its phase formula, and we use $\psi[\psi_{ph} \triangleleft \xi]$ for the formula that results from ψ by replacing ψ_{ph} with ξ. Since ψ_{ph} occurs only once in ψ, we use the fact that ψ_{ph} appears with a single polarity. We present here the construction for positive polarity which is the case in all practical formulas (the case for negative polarity is analogous).

The algorithm has two parts. First, we generate the *stuttering* Kripke structure $\mathcal{K}^{st}$ whose paths are the stuttering expansions of paths in the original Kripke structure $\mathcal{K}$. Then, we modify the admissible formula ψ into ψ_{sync} such that $\mathcal{K} \models \forall\pi_1 \dots \pi_n.\mathsf{E}.\psi$ if and only if $\mathcal{K}^{st} \models \forall\pi_1 \dots \pi_n.\psi_{sync}$. We describe each of the concepts separately.

Phase Formulas. We first define *atomic phase formulas* ($\bigwedge_{p\in P} p_{\pi_i} \leftrightarrow p_{\pi_j}$) which are characterized by (π_i, π_j, P), where $P \subseteq \mathsf{AP}$ and π_i and π_j are two different trace variables. We use *color* to refer to a valuation of the variables in P. Essentially, an atomic phase formula asserts that all propositions in P coincide in both traces at all points in time, that is, both traces exhibit the same sequence of colors. Since the passage of time proceeds at different speeds in the different traces—according to the trajectory—atomic phase formulas state the traces for π_i and π_j are sequences of phases of the same color, where corresponding phases may have different lengths. A phase formula is formed from atomic formulas as follows:

$$\square\Big(\bigwedge_{p\in P^1} p_{\pi_i^1} \leftrightarrow p_{\pi_j^1} \wedge \dots \wedge \bigwedge_{p\in P^k} p_{\pi_i^k} \leftrightarrow p_{\pi_j^k}\Big)$$

We use $\mathcal{P} : \{(\pi_i^1, \pi_j^1, P^1), \dots, (\pi_i^k, \pi_j^k, P^k)\}$ for the collection of predicates and trace variables that characterize a phase formula.

Stuttering Kripke Structure. We start from $\mathcal{K}$ and create $\mathcal{K}^{st}$ that accepts the stuttering expansions of traces in $\mathcal{K}$. First, the alphabet of atomic propositions is enriched with a fresh proposition st, that is $\mathsf{AP}^{st} = \mathsf{AP} \cup \{st\}$, to encode whether the state represents a real move or a stuttering move. Given $\mathcal{K} = \langle S, S_{init}, \delta, L\rangle$, the stuttering Kripke structure is $\mathcal{K}^{st} = \langle S^{st}, S_{init}, \delta^{st}, L^{st}\rangle$ where:

- $S^{st} = S \cup \{s^{st} \mid s \in S\}$ contains two copies of each state in S, where we use s^{st} to denote the stuttering state that corresponds to s;
- $\delta^{st} = \delta \cup \{(s, s^{st})\} \cup \{(s^{st}, s^{st})\} \cup \{(s^{st}, s') \mid \text{ for every } (s, s') \in \delta\}$.
- $L^{st}(s) = L(s)$ for $s \in S$, and $L^{st}(s^{st}) = L(s) \cup \{st\}$.

The construction generates a Kripke structure $\mathcal{K}^{st}$ which is linear in the size of the original Kripke structure $\mathcal{K}$. It is easy to see that every stuttering expansion of a path of $\mathcal{K}$ has a corresponding path in $\mathcal{K}^{st}$, where the repeated version of state s is captured by state s^{st}. Conversely every path ρ' in $\mathcal{K}^{st}$ whose trace satisfies $\square\diamond\neg st$ can be turned into its "stuttering compression" by removing all stuttering states, which is a path of $\mathcal{K}$. Note that the constraint $\square\diamond\neg st$ guarantees that there are infinitely many non-stuttering positions in ρ', so ρ is well-defined. Hence, this constructions provides a one-to-one correspondence between a trajectory toguether with a tuple of traces of $\mathcal{K}$, and the corresponding tuple of traces of $\mathcal{K}^{st}$.

State and Monadic Formulas are not Affected by Trajectories. State formulas are relational formulas that are evaluated at the beginning of the computation. Temporal monadic formulas only refer to one trace variable and are stuttering invariant by definition. Therefore, none of these formulas are affected by the stuttering induced by a trajectory, as the relative stuttering among traces does not affect their truth valuation. We first note that given a trace assigned for each of the trace variables in $\mathsf{Vars}(\varphi)$ the truth value of state formulas and monadic formulas does not depend on the trajectory chosen.

Phase Alignment of Asynchronous Sequences. We use the stuttering in $\mathcal{K}^{st}$ to encode the relative progress of traces as dictated by a trajectory. We will now introduce synchronous HyperLTL formulas to reason in $\mathcal{K}^{st}$ about the corresponding states during the asynchronous evaluation in $\mathcal{K}$. The important concept is that of "phase changes", which are the points in a trace σ at which the valuation of the predicates P in an atomic phase formula (π_i, π_j, P) change. Let Π be a trace assignment for traces in $\mathcal{K}$ that maps π_i to a pointed trace (σ, l). We say that in assignment Π, trace variable π_i is *about to change phase* with respect to (π_i, π_j, P) if for some $p \in P$ either $p \in \sigma(l)$ but $p \notin \sigma(l+1)$ or $p \notin \sigma(l)$ but $p \in \sigma(l+1)$. Note that in $\mathcal{K}^{st}$ the next relevant letter (the one corresponding to $\sigma(l+1)$ is the first letter that is not a stuttering letter). Formula $change_P(\pi_i)$ captures that the next non-stuttering step of π_i is a phase change (with respect to predicates in P and therefore with respect to atomic phase formula α):

$$change_P(\pi_i) \stackrel{\text{def}}{=} \bigvee_{p \in P} p_{\pi_i} \not\leftrightarrow \bigcirc(st_{\pi_i} \,\mathcal{U}\, p_{\pi_i})$$

A phase change for π_i in atomic phase formula (π_i, π_j, P) implies that π_j must also proceed to change phase. The second observation is that when π_i and π_j are not changing phases, any choice that the trajectory makes will preserve the valuation of the atomic phase formula.

We now capture formally this intuition as formulas. Predicate $move(\pi_i) \stackrel{\text{def}}{=} \bigcirc(\neg st_{\pi_i})$ indicates whether trace variable π_i will move (and not stutter) at a given instant of the computation. The following temporal formula captures the consistency criteria of phase changes as a synchronized decision for moving traces π_i and π_j related by an atomic phase formula (π_i, π_j, P):

$$align_{(\pi_i,\pi_j,P)} \stackrel{\text{def}}{=} \begin{pmatrix} (move(\pi_i) \wedge move(\pi_j)) \rightarrow (change_P(\pi_i) \leftrightarrow change_P(\pi_j)) \wedge \\ (move(\pi_i) \wedge \neg move(\pi_j)) \rightarrow \neg change_P(\pi_i) \qquad\qquad \wedge \\ (\neg move(\pi_i) \wedge move(\pi_j)) \rightarrow \neg change_P(\pi_j) \end{pmatrix}$$

We will reduce the model-checking problem in A-HLTL to checking in $\mathcal{K}^{st}$ that tuples of traces that align phase changes—for all atomic phase formulas— satisfy all sub-formulas of the specification ψ. The following two formulas express that all atomic phase formulas align, and that all traces are fair (all traces eventually move):

$$phase \stackrel{\text{def}}{=} \bigwedge_{(\pi_i,\pi_j,P)\in\mathcal{P}} align_{(\pi_i,\pi_j,P)} \qquad\qquad fair \stackrel{\text{def}}{=} \bigwedge_{\pi_i\in\{\pi_1\ldots\pi_n\}} \Box\Diamond\neg st_i$$

We will then check in $\mathcal{K}^{st}$ that all stuttering traces that align phases and are fair satisfy the desired formula ψ, that is $(\Box phase \wedge fair) \rightarrow \psi$. Note that all those tuples of traces that do not align phases are ruled out in the antecedent.

A final technical detail in the construction is that we must guarantee that for all tuples of paths of $\mathcal{K}$ there are stuttering expansions that are fair and align phases, and that they have the same number of phases. Otherwise, there are paths of $\mathcal{K}$ that cannot be aligned, which inevitably leads to a violation of ψ_{ph}. It could be the case that some tuple of traces of $\mathcal{K}$ cannot possibly align the phase changes corresponding to all atomic phase formulas. This can happen in two cases: (1) when two traces have different number of phases, and (2) when there is a circular dependency between the atomic formulas that force the trajectory to synchronize the traces in incompatible orders. The first case is captured by:

$$missalign \stackrel{\text{def}}{=} \bigvee_{(\pi_i,\pi_j,P)} \Big(\Box\neg change_P(\pi_i)\Big) \not\leftrightarrow \Big(\Box\neg change_P(\pi_j)\Big)$$

The second case is captured by the following formula, where $cycles(\psi_{ph})$ are the sequences of atomic formulas that form a simple cycle, that is $[(\pi^0,\pi^1,P^0),(\pi^1,\pi^2,P^1)\ldots(\pi^k,\pi^0,P^k)]$ such that the second trace variable is the first trace variable of the next atomic phase formula, circularly (see Ex. 1 below):

$$block \stackrel{\text{def}}{=} \bigvee_{C\in cycles(\psi_{ph})} \Big(\bigwedge_{(\pi_i,\pi_j,P)\in C} change_P(\pi_i) \wedge \neg change_P(\pi_j)\Big)$$

Essentially, *block* encodes whether the set of traces involved cannot proceed without violating *phase*, because *align* forbids all traces involved to move. Hence, the formula $\mathit{phase}\,\mathcal{U}\,(\mathit{missalign} \vee \mathit{block})$ captures to those traces of $\mathcal{K}^{st}$ that contain an aligned prefix of computation that lead to a miss-alignment or a block. The proof of correctness shows that given a tuple of traces of $\mathcal{K}$, if there is a trajectory that aligns the phase changes (which must exist if there is a trajectory that makes ψ_{ph} true), then all trajectories that respect $\Box\mathit{phase}$ will also align the phase changes (and also satisfy ψ_{ph}).

We are finally ready to describe the synchronous phase formula ψ_{sync}. First, this formula is only evaluated against tuples of fair traces, which correspond to the stuttering extensions of paths of $\mathcal{K}$. Then, the phase formula ψ_{ph} is translated into a formula that captures (1) that following a phase alignment cannot lead to a block or to two traces changing phases a different number of times, and (2) that if phases are aligned then ψ_{ph} holds. Formally,

$$\psi_{sync} \stackrel{\text{def}}{=} \mathit{fair} \rightarrow \psi[\psi_{ph} \triangleleft \psi'], \qquad \text{where } \psi' = \begin{pmatrix} \neg(\mathit{phase}\ \mathcal{U}\ (\mathit{missalign} \vee \mathit{block})) \ \wedge \\ \Box\mathit{phase} \rightarrow \psi_{ph} \end{pmatrix}$$

Example 1. We illustrate the previous definitions with the Kripke structures $\mathcal{K}_1$, $\mathcal{K}_2$ and $\mathcal{K}_3$ in Fig. 5 and their stuttering variants $\mathcal{K}_1^{st}$, $\mathcal{K}_2^{st}$ and $\mathcal{K}_3^{st}$ Consider formula $\forall\pi_1.\forall\pi_2.\mathsf{E}.\Box(a_{\pi_1} \leftrightarrow a_{\pi_2})$. Consider the following trace assignments:

$\Pi^1(\pi_1) \mapsto \{\}\ \{st\}\ \{a\}\ \ldots$	$\Pi^2(\pi_1) \mapsto \{\}\ \{a\}\ \{a\}\ \ldots$	$\Pi^3(\pi_1) \mapsto \{a\}\ \{\}\ \{\}\quad \ldots$
$\Pi^1(\pi_2) \mapsto \{\}\ \{\}\quad \{a\}\ \ldots$	$\Pi^2(\pi_2) \mapsto \{\}\ \{\}\quad \{a\}\ \ldots$	$\Pi^3(\pi_2) \mapsto \{a\}\ \{\}\ \{a\}\ \ldots$

Consider the trace assignment Π^1 on the left, where π_1 is a trace of $\mathcal{K}_1^{st}$ corresponding to the path of $\mathcal{K}_1$ that visits s_1, and π_2 corresponds to the path that visits s_2. This trace assignment aligns the atomic phase formula $(\pi_1, \pi_2, \{a\})$ at all positions. In particular, at position 0, we have $\mathit{change}_{\{a\}}(\pi_1)$, but $\neg\mathit{change}_{\{a\}}(\pi_2)$, and $\neg\mathit{move}(\pi_1)$ and $\mathit{move}(\pi_2)$, as $\mathit{align}_{\{a\}}$ requires.

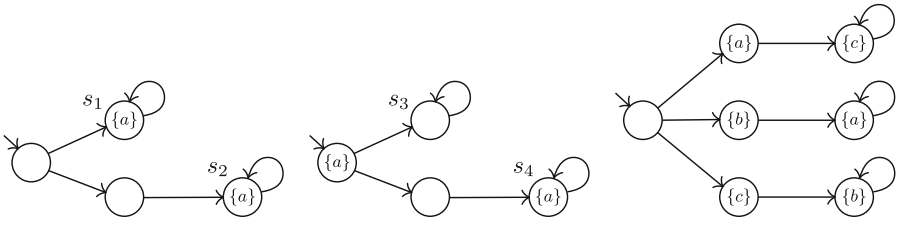


Fig. 5. Kripke structure $\mathcal{K}_1$ (left), $\mathcal{K}_2$ (middle) and $\mathcal{K}_3$ (right).

$\Pi(\pi_1) \mapsto \{\}\ \{\}\ \{a\}\ \{c\}\dots$
$\Pi(\pi_2) \mapsto \{\}\ \{\}\ \{b\}\ \{a\}\dots$
$\Pi(\pi_3) \mapsto \{\}\ \{\}\ \{c\}\ \{b\}\dots$

Consider now the trace assignment Π^2 in the middle, where again π_1 corresponds to the path in $\mathcal{K}_1^{st}$ that visits s_1 and π_2 the path that visits s_2. In this case, we have $\neg \mathit{align}_{\{a\}}$ at position 0 because $\mathit{change}_{\{a\}}(\pi_1)$ and $\neg \mathit{change}_{\{a\}}(\pi_2)$ hold, and both $\mathit{move}(\pi_1)$ and $\mathit{move}(\pi_2)$. Consider now Π^3 on the right, where π_1 corresponds to the path of $\mathcal{K}_2^{st}$ that visits s_3 and π_2 to the path of $\mathcal{K}_2^{st}$ that visits s_4. In this case $\mathit{align}_{\{a\}}$ holds at 0 and *missalign* holds at 1 because at 1, $\Box\neg \mathit{change}_{\{a\}}(\pi_1)$ holds, but not $\Box\neg \mathit{change}_{\{a\}}(\pi_2)$. Therefore, $\mathit{phase}\,\mathcal{U}\,(\mathit{missalign} \vee \mathit{block})$ holds for Π^3. Finally, consider $\forall\pi_1.\forall\pi_2.\forall\pi_3.\mathsf{E}.\Box(a_{\pi_1} \leftrightarrow a_{\pi_2} \wedge b_{\pi_2} \leftrightarrow b_{\pi_3} \wedge c_{\pi_3} \leftrightarrow c_{\pi_2})$ and the trace assignment Π of $\mathcal{K}_3^{st}$ shown below on the left. In this case *phase* holds at position 0 and *block* holds at position 1. This is because $\mathit{change}_{\{a\}}(\pi_1)$ and $\neg \mathit{change}_{\{a\}}(\pi_2)$, $\mathit{change}_{\{b\}}(\pi_2)$ and $\neg \mathit{change}_{\{b\}}(\pi_3)$, and $\mathit{change}_{\{c\}}(\pi_3)$ and also $\neg \mathit{change}_{\{c\}}(\pi_1)$. This illustrates that it will not be possible to align all three atomic phase formulas.

We are now ready to state the main result of this section.

Theorem 1. *Let $\mathcal{K}$ be a Kripke structure and ψ an admissible formula. Then, $\mathcal{K} \models \forall\pi_1 \dots \pi_n.\mathsf{E}.\psi$ if and only if $\mathcal{K}^{st} \models \forall\pi_1 \dots \pi_n.\psi_{sync}$.*

Dually, to show that the $\exists^*$ fragment is decidable, we consider replacing ψ_{ph} by the formula

$$\psi_{esync} \stackrel{\text{def}}{=} \mathit{fair} \wedge \psi[\psi_{ph} \triangleleft (\Box \mathit{phase} \wedge \psi_{ph})]$$

Theorem 2. *Let $\mathcal{K}$ be a Kripke structure and ψ an admissible formula. Then $\mathcal{K} \models \exists\pi_1 \dots \pi_n.\mathsf{E}.\psi$ if and only if $\mathcal{K}^{st} \models \exists\pi_1 \dots \pi_n.\psi_{esync}$.*

The proof of Theorem 2 takes a witness tuple and trajectory in $\mathcal{K}$ and shows that the induced tuple in $\mathcal{K}^{st}$ is *fair*, satisfies $\Box \mathit{phase}$ and that the valuation of ψ_{ph} is preserved. Similarly, as before, tuples of traces of $\mathcal{K}^{st}$ that are fair and follow phase alignments induce a trajectory on their stuttering compression that also preserve ψ_{ph}.

Corollary 1. *The problems of model-checking $\forall^*$ admissible* A-HLTL *formulas and $\exists^*$ admissible* A-HLTL *formulas is decidable.*

We finally consider the negation of phase formulas, called *co-phase formulas*, which are formulas of the form $\Diamond\neg R$ where R a conjunction of atomic phase formulas. Interestingly, deciding co-admissible formulas (consisting of Boolean combinations of state-formulas, monadic temporal formulas and one co-phase formula in positive polarity) is easier than before, as one can turn the co-phase formula into a monadic formula enumerating all the violations of the atomic phase formulas ($p \in P$ such that $p_{\pi_i} \not\leftrightarrow p_{\pi_j}$) turns the atomic phase formula into $(\Diamond p_{\pi_i} \wedge \Diamond\neg p_{\pi_j}) \vee (\Diamond\neg p_{\pi_i} \wedge \Diamond p_{\pi_j})$. It follows that model-checking co-admissible formulas is also decidable (for both $\forall^*$ and $\exists^*$). Note that an admissible formula

in negative polarity is a co-admissible formula in positive polarity (and vice versa). Finally, since $\mathcal{K} \models \forall \pi_1 \ldots \forall \pi_n.\mathsf{A}.\psi$ if and only if $\mathcal{K} \not\models \exists \pi_1 \ldots \exists \pi_n.\mathsf{E}.\neg\psi$, it follows that model-checking is also decidable for the A modality for both admissible and co-admissible formulas (in both polarities), and for both the $\forall^*$ and $\exists^*$ fragments.

Theorem 3. *Model-checking $\forall^*$ or $\exists^*$ admissible and co-admissible formulas is decidable both for formulas with E and formulas with A.*

4.2 The Accelerating Construction

The admissible formula in the stuttering construction can express many formulas of interest, but the quantifier structure admits no quantifier alternation. We now consider a second decidable fragment for A-HLTL formulas consisting of formulas with arbitrary quantification $\mathbb{Q}_1\pi_1.\mathbb{Q}_2\pi_2.\ldots.\mathbb{Q}_n\pi_n\mathsf{E}.\psi$ such that $\mathbb{Q}_i \in \{\forall, \exists\}$, but where ψ is an admissible formula where all atomic phase formulas use the same atomic predicates $P \subseteq \mathsf{AP}$. We call these admissible formulas *simple admissible formulas*. The proof of decidability proceeds this time by creating the *accelerated* Kripke structure $\mathcal{K}^{acc}$, where paths jump in one step to the next phase change, and reducing to a HyperLTL model-checking problem on $\mathcal{K}^{acc}$.

Accelerated Kripke Structure. The main idea of the acceleration construction is to convert a finite sequence of transitions in $\mathcal{K}$ that only change phase in the last transition into a single transition in $\mathcal{K}^{acc}$. Also, an infinite sequence of transitions with no phase change is transformed into a self-loop around a sink state. The alphabet remains the same, AP. Given $\mathcal{K} = \langle S, S_{init}, \delta, L\rangle$, the accelerated Kripke structure is $\mathcal{K}^{acc} = \langle S^{acc}, S_{init}, \delta^{acc}, L^{acc}\rangle$ where:

- $S^{acc} = S \cup \{s_\perp \mid s \in S\}$ contains two copies of each state in S, where we use $s_\perp$ to denote the sink state associated with s. We use $color(s)$ for the phase of s, that is, the concrete valuation in s of the Boolean predicates in P of the atomic phase formula.
- For every states $s, s' \in S$ such that $color(s) \neq color(s')$, if there is a finite path $ss_2s_3 \ldots s_n s'$ in $\mathcal{K}$ such that $color(s) = color(s_2) = \cdots = color(s_n)$, then we add a transition (s, s') to δ^{acc}. These transitions model the jump at the frontier of phase changes. Additionally, if s can be a sink we add a transition $(s, s_\perp)$ and a self-loop from $s_\perp$ to itself.
- $L^{acc}(s) = L(s)$ for $s \in S$, and $L^{acc}(s_\perp) = L(s)$.

This construction can, with standard techniques, be enriched to encode the satisfaction of the temporal monadic formulas along paths of $\mathcal{K}$, and then also accelerate the fairness conditions (annotating the accepting states reached along the accelerated paths) into $\mathcal{K}^{acc}$.

Relating Paths to Accelerated Paths. We now define two auxiliary functions to aid in the proof.

- The first function, *acc*, maps paths in $\mathcal{K}$ into paths in $\mathcal{K}^{acc}$. Let s be an arbitrary state of $\mathcal{K}$ and $\rho : ss_1s_2s_3\ldots$ an outgoing path from s. Either there are infinitely many phase changes in ρ or only finitely many changes. We create the path $\rho' = acc(\rho)$ as follows. The initial state of ρ, that is, s, is preserved. The states s_{i_j} in σ that are color changes (that is $color(s_{i_j-1}) \neq color(s_{i_j})$ are also preserved, while the states s_k with $color(s_{k-1}) = color(s_k)$ are removed from ρ. If there are only finitely many color changes in ρ, with r being the last state preserved, then we pad the path with $r_\perp^\omega$, so ρ' is also an infinite path. It is easy to see that ρ' is a path of $\mathcal{K}^{acc}$ outgoing s. It is also easy to see that the phase changes in ρ and ρ' are the same.
- The second map, *dec*, takes a path $\rho' : ss'_1s'_2\ldots$ of $\mathcal{K}^{acc}$ and maps it to a path of $\mathcal{K}$ as follows. For every transition (s'_i, s'_{i+1}) in ρ such that s'_{i+1} is not of the form $r_\perp$, there is a finite path $r_1r_2\ldots r_m$ in $\mathcal{K}$ from s'_i into s'_{i+1} that visits only states with the same color as s'_i, except s_{i+1} that is a color change. In ρ, we insert $r_1r_2\ldots r_m$ between s'_i and s'_{i+1}. Now, if for some j, s'_j is of the form $r_\perp$ then $s'_k = r_\perp$ for all $k > j$. In $\mathcal{K}$ there must an infinite path from s'_j that only visits the same color as s'_j. We remove all successor states after the first such $r_\perp$ state and replace it with one such infinite path.

Given a trace assignment Π for formula $\mathbb{Q}_1\pi_1.\ldots.\mathbb{Q}_n\pi_n.\mathsf{E}.\psi$ that assigns $\Pi(\pi_i) = (\sigma_i, 0)$ for every i and a path assignment Π' for formula $\mathbb{Q}_1\pi_1.\ldots.\mathbb{Q}_n\pi_n.\psi$ that assigns $\Pi'(\pi_)i = (\sigma'_i, 0)$, we write $acc(\Pi) = \Pi'$ if the paths that generate the corresponding traces are related by *acc*. Similarly we defined $dec(\Pi') = \Pi$. It is easy to show from the construction above that if $\Pi \models \mathsf{E}\psi$ then $acc(\Pi) \models \psi$, and if $\Pi' \models \psi$ then $dec(\Pi') \models \mathsf{E}\psi$.

The main result for the accelerating construction follows immediately from this observation and allows to reduce the model-checking problem to HyperLTL.

Theorem 4. *Let $\mathcal{K}$ be an arbitrary Kripke structure, $\mathbb{Q}_1\pi_1.\ldots.\mathbb{Q}_n\pi_n.\mathsf{E}.\psi$ such that ψ is a simple admissible formula. Then $\mathcal{K} \models \mathbb{Q}_1\pi_1.\ldots.\mathbb{Q}_n\pi_n\mathsf{E}.\psi$ if and only if $\mathcal{K}^{acc} \models \mathbb{Q}_1\pi_1.\ldots.\mathbb{Q}_n\pi_n.\psi$.*

4.3 Decidable Practical A-HLTL Formulas

We revisit the properties expressed in Sect. 3.2.

- *Linearizability.* The property φ_{LNZ} is of the form $\forall\pi.\exists\pi'.\mathsf{E}.\square(\mathsf{history}_\pi \leftrightarrow \mathsf{history}_{\pi'})$ where the temporal formula is a simple admissible formula. Therefore φ_{LNZ} is decidable by the accelerating construction.
- *Goguen and Meseguer's non-interference.* The property φ_{GMNI} is expressed by $\forall\pi.\exists\pi'.\mathsf{E}.(\square\lambda_{\pi'}) \wedge \square(lo_\pi \leftrightarrow lo_{\pi'})$, that is, a Boolean combination of a monadic temporal formula and a simple admissible formula. Therefore, φ_{GMNI} is decidable by the acceleration algorithm.
- *Not never terminates.* Formula φ_{NNT} is simply a Boolean combination of state formulas and monadic temporal formulas: $\forall\pi.\exists\pi'.\exists\pi''.\mathsf{E}.(\pi[0] = \pi'[0] = \pi''[0]) \rightarrow (\diamond\,\mathsf{term}_{\pi'} \wedge \square\neg\mathsf{term}_{\pi''})$, so it is again decidable by the acceleration construction.

– *Termination-insensitive noninterference.* To handle φ_{TIN} we rewrite the formula as follows

$$\varphi_{\mathsf{TIN}} \stackrel{\text{def}}{=} \forall\pi.\forall\pi'.\mathsf{E}.\Big(l_\pi \leftrightarrow l_{\pi'}\Big) \rightarrow \begin{pmatrix}(\Box\neg\mathsf{term}_\pi \vee \Box\neg\mathsf{term}_{\pi'}) \vee \\ \Box\big((l_\pi \wedge \mathsf{term}_\pi) \leftrightarrow (l_{\pi'} \wedge \mathsf{term}_{\pi'})\big)\end{pmatrix}$$

Note that $(l_\pi \wedge \mathsf{term}_\pi)$ can be turned into a state predicate of π. This formula is equivalent because the last case is evaluates precisely to $l_\pi \leftrightarrow l_{\pi'}$ when both traces terminate. This formula can be handled by the stuttering construction.

– *Termination-sensitive noninterference.* Similarly, to handle φ_{TSN} we rewrite the formula as

$$\varphi_{\mathsf{TSN}} \stackrel{\text{def}}{=} \forall\pi.\forall\pi'.\mathsf{E}.\Big(l_\pi \leftrightarrow l_{\pi'}\Big) \rightarrow \begin{pmatrix}(\Box\neg\mathsf{term}_\pi \wedge \Box\neg\mathsf{term}_{\pi'}) \vee \\ \Box\big((l_\pi \wedge \mathsf{term}_\pi) \leftrightarrow (l_{\pi'} \wedge \mathsf{term}_{\pi'})\big)\end{pmatrix}$$

This is again equivalent because the last case again is the only relevant case when both paths terminate. Again, this case is covered by the stuttering construction.

5 Undecidability and Lower-Bound Complexity

In this section, we show that the general problem of model-checking A-HLTL is undecidable. Then, we show a polynomial reduction from the synchronous HyperLTL model-checking into A-HLTL model-checking, which shows that even for those A-HLTL formulas for which the model-checking is decidable, this problem is no easier than the corresponding problem for HyperLTL, which is known to be PSPACE-hard in the size of the Kripke structure.

Theorem 5. *Let $\mathcal{K}$ be a Kripke structure and φ be an asynchronous HyperLTL formula. The problem of determining whether or not $\mathcal{K} \models \varphi$ is undecidable.*

Proof (sketch). We reduce the complement of the *post correspondence problem (PCP)* [23,26] to the A-HLTL model checking problem. PCP consists of a set of dominos, for example, of the form $[\frac{w}{v}] = \{[\frac{b}{ca}], [\frac{a}{ab}], [\frac{ca}{a}], [\frac{abc}{c}]\}$ and the problem is to decide whether there is a sequence of dominos (with possible repetitions), such that the upper and lower finite strings of the dominos are equal. A solution to the above set of dominos is the sequence $[\frac{a}{ab}][\frac{b}{ca}][\frac{ca}{a}][\frac{a}{ab}][\frac{abc}{c}]$. We map a given set of dominos to a Kripke structure that allows arranging the dominos in a sequence (see Fig. 6 for an example), where v and w indicate lower and upper words, respectively, dom^i is for each domino $[\frac{w_i}{v_i}]$, and proposition lc marks whether or not a new letter is processed. The A-HLTL formula in our reduction is the following such that $dom_{\pi_w} \stackrel{\text{def}}{=} \bigvee_{i\in[1..k]} dom^i_{\pi_w}$:

$$\varphi_{\overline{pcp}} \stackrel{\text{def}}{=} \forall\pi_w\forall\pi_v.\mathsf{E}.\Big(\varphi_{type} \rightarrow (\varphi_{domino} \vee \varphi_{word})\Big)$$

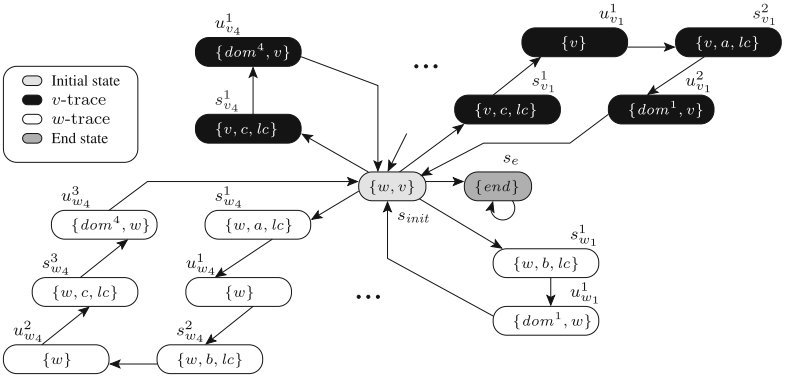


Fig. 6. Mapping from PCP to model checking A-HLTL (only construction for dominos $[\frac{w_1}{v_1}] = [\frac{b}{ca}]$ and $[\frac{w_4}{v_4}] = [\frac{abc}{c}]$ are shown).

$$\text{where} \quad \varphi_{type} \stackrel{\text{def}}{=} \Big((w_{\pi_w} \wedge \neg v_{\pi_w})\, \mathcal{U}\ end_{\pi_w}\Big) \wedge \Big((\neg w_{\pi_v} \wedge v_{\pi_v})\, \mathcal{U}\ end_{\pi_v}\Big)$$

$$\varphi_{domino} \stackrel{\text{def}}{=} \square(dom_{\pi_w} \leftrightarrow dom_{\pi_v}) \wedge \Diamond \bigvee_{i=1}^{k} dom^i_{\pi_w} \not\leftrightarrow dom^i_{\pi_v}$$

$$\varphi_{word} \stackrel{\text{def}}{=} \square(lc_{\pi_w} \leftrightarrow lc_{\pi_v}) \quad \wedge \Diamond \bigvee_{l \in \Sigma_{pcp}} (l_{\pi_w} \not\leftrightarrow l_{\pi_v})$$

The intention of formula $\varphi_{\overline{pcp}}$ is that the Kripke structure is a model of the formula if and only if the original PCP problem has *no* solution. Intuitively, formula φ_{type} forces trace π_w (respectively, π_v) to traverse only the traces labeled by w (respectively, v) to build a w-word (respectively, v-word). Formula φ_{domino} establishes that the trajectory aligns the positions at which the domino indices are checked and at last once the index is different. Finally, formula φ_{word} captures if π_w and π_v are aligned to compare the letters, at least one pair of the letters prescribed by the existential trajectory are different. In the detailed proof in [4], we show that the constructed Kripke structure satisfies formula $\varphi_{\overline{pcp}}$ if and only if the answer to deciding PCP is negative. □

Theorem 5 above implies that there is no algorithm to decide the model-checking problem correctly for every formula and every system. However, as we saw in Sect. 4 for some formulas the model-checking problem is decidable. We now show that in these cases the problem is at least as hard as model-checking HyperLTL, which is known to be PSPACE-hard [7,24].

Theorem 6. *Given a HyperLTL formula φ and a Kripke structure $\mathcal{K}$ there is a* A-HLTL *formula φ' and a Kripke structure $\mathcal{K}'$ such that $\mathcal{K}'$ is linear in the size of $\mathcal{K}$, φ' is polynomial on the size of φ and $\mathcal{K} \models \varphi$ if and only if $\mathcal{K}' \models \varphi'$.*

The proof proceeds as follow. Giving $\mathcal{K}$ we build a Kripke structure $\mathcal{K}'$ that alternates between real states in $\mathcal{K}$ and synchronization states. Then the formula

is transformed to force alternations at every other step, therefore forcing the trajectory to synchronize (see [4] for details). Since the model-checking problem for HyperLTL is PSPACE-hard on the size of the Kripke structure, the same follows for A-HLTL.

Corollary 2. *For asynchronous HyperLTL formulas, the model checking problem is PSPACE-hard in the size of the system.*

6 Case Studies and Evaluation

We applied our algorithm for the $\forall^*_\pi \mathsf{E}$ A-HLTL fragment to several examples. After manually reducing the asynchronous model checking problem to a synchronous one, we use MCHyper [10,11] to check our property. MCHyper is a model checker for synchronous HyperLTL that can handle formulas with up to one quantifier alternation. It computes the self composition of the system and composes it with the formula automaton. ABC [6] is then used as the backend tool checking the reachability of a violation.

Our reduction from the asynchronous to the synchronous semantics follows the stuttering construction described in Sect. 4.1. To model check a system against an A-HLTL formula, we first add a stuttering input to the system that forces the system to stutter in the current state. The transformed formula ensures that the stuttering guarantees synchronous phase changes. In future work, we will fully automate our reduction resulting in a verification tool for asynchronous hyperproperties from the decidable fragment. We now describe the various case studies[1]. All our experiments were performed on a MacBook Pro with a 3.3 GHz processor and 16 GB of RAM running MacOS 11.1.

6.1 Compiler Optimizations

We modeled the source and target programs of different compiler optimization techniques (from [20]) as finite state machines encoded as circuits, and used asynchronous hyperproperties to prove the correspondence between both programs. We analyzed the following optimizations:

- Common Branch Factorization (CBF), where expressions occurring in both branches of a conditional are factored out;
- Loop Peeling (LP), which consists in unrolling of a loop that is executed at least once;
- Dead Branch Elimination (DBE), that is, removing conditional checks and their branches that are unreachable; and
- Expression Flatting (EF), which splits complex computations into several explicit steps.

[1] The experimental data is publicly available at https://github.com/reactive-systems/MCHyper in `case-studies/asynchronous-hyperltl_2021`.

Table 1. Verification times of MCHyper and system sizes in number of latches (#ls) and and-gates (#ands) for the case studies.

Optimizations	System Size		Time (s)
	#ls	#ands	
EF	12	64	0.6
DBE	16	128	0.8
CBF	16	145	2.7
LP	28	514	365.9
CBF+DBE	16	137	11.4
CBF+DBE+EF	20	175	10.0
CBF+EF	20	180	1.7
EF+LP	41	8642	1315.2

(a) Compiler Optimizations

Propery	System Size		Time (s)
	#ls	#ands	
SPI-correct	30	175	65.7
SPI-term	33	296	155.8

(b) SPI

Besides evaluating each optimization individually, we also examined several combinations of these optimizations. Each optimization affects the alignment between source and target program, so synchronous hyperproperties fail to recognize the correspondence between both programs. Using asynchronous hyperproperties instead allows us to compensate for this misalignment by stuttering the programs accordingly. Essentially, each optimization is checked against the following A-HLTL formula in which π represents traces from the source program and π' traces from the target program:

$$\forall\pi.\forall\pi'.\mathsf{E}.(\bigwedge_{i\in I} i_\pi \leftrightarrow i_{\pi'}) \rightarrow (\square \bigwedge_{o\in O} o_\pi \leftrightarrow o_{\pi'})$$

This formula states that for all pairs of traces that initially agree on the inputs from the set I there exists a trajectory that aligns the phase changes of the outputs in set O. We use the stuttering construction and MCHyper to verify that in all cases the source and target programs go through the same phases of possibly different length. The results of this case study are summarized in Table 1(a). We note that A-HLTL model-checking subsumes the approach in [20] based on construction of a *buffer automaton* to reason about the alignment of executions.

6.2 SPI Bus Protocol

The Serial Peripheral Interface (SPI) is a bus protocol that supports a single main component's communication with multiple secondary components. Each secondary can be selected individually by the main via the secondary's own *ss* ("secondary select") input signal. If a secondary is enabled (that is, if $\neg ss$ holds as the secondary select is "active low"), it reads the *mosi* (main out, secondary in) signal and writes to the *miso* (main in, secondary out) wire.

We verify the behavior of a single SPI secondary component that receives an input which it sends to the main component upon request. This behavior should always be the same, independent of when the secondary is enabled or how fast the bus protocol's "serial clock" (*sclk*) set by the main component ticks compared to the secondary's internal clock. The A-HLTL formula we check is the following (see observational determinism in Sect. 1):

$$\forall\pi.\forall\pi'.\mathsf{E}.\left(\begin{array}{c}\bigwedge\limits_{i\in\{in,init\}} i_\pi \leftrightarrow i_{\pi'}\\ \wedge\\ \textit{SPI input assumptions}\end{array}\right) \rightarrow \square\left(\begin{array}{c}(miso_\pi \wedge \neg sclk_\pi \wedge \neg ss_\pi)\\ \leftrightarrow\\ (miso_{\pi'} \wedge \neg sclk_{\pi'} \wedge \neg ss_{\pi'})\end{array}\right)$$

This formula (called SPI-correct in Table 1(b)) ensures that for all pairs of traces π and π' that agree on the initial configuration, on the input, and additional *SPI input assumptions*, there is a trajectory that aligns their relevant behavior. We consider it relevant that both secondaries agree on their *miso* output whenever they are enabled and the *sclk* is low. Checking *miso* only when the *sclk* is low is sufficient as changes on *miso* only occur at falling edges of the *sclk*. The *SPI input assumptions* are required to guarantee the implicit assumptions of the protocol, for example, that the *sclk* behaves as an infinitely ticking clock. By introducing additional variables and applying logical transformations, we obtain an equivalent formula that syntactically lies in the fragment of the stuttering construction. Again, we reduce this model checking problem to the synchronous semantics and use MCHyper to perform the verification.

In a second experiment, we modified the system to send the value only once and checked it for termination insensitive noninterference SPI-term (see Sects. 3.2 and 4.3). In our setup, we use the variable *term* to flag that the secondary has sent the full value. In the premise of the formula, we require that the input value is equal on both traces and again assume that the inputs conform to the SPI protocol. The conclusion checks if both secondaries have sent the same values by using additional variables that are set together with *term*. The results of this case study are summarized in Table 1(b).

7 Related Work

The study of specific hyperproperties, such as noninterference, dates back to the seminal work by Goguen and Meseguer [14] in the 1980s. The first systematic study of hyperproperties is due to Clarkson and Schneider [8].

It is well-known that classic specification languages like LTL cannot express hyperproperties. There are two principal methods with which the standard logics have been extended to express hyperproperties:

– The first method is the quantification over variables that identify specific paths or traces. The temporal logics LTL, CTL* have been extended with quantification over traces and paths, resulting in the temporal logics HyperLTL and HyperCTL* [7]. There are also extensions of the μ-calculus, most

recently, the temporal fixpoint calculus H_μ [15], which extends the linear time μ-calculus [3] with path quantifiers and indexed next operators.
– The second method is the addition of the equal-level predicate E to first-order and second-order logics, like MPL, MSO, FOL, and S1S, which results in the logics FOL[E], S1S[E], MPL[E], MSO[E] [9,13].

HyperCTL*, MPL[E], and MSO[E] are branching-time logics, we therefore focus in the following on the linear-time logics HyperLTL, H_μ, FOL[E], and S1S[E]. Among these logics, HyperLTL is the only logic for which practical model-checking algorithms are known [10,11,17]. For HyperLTL, the algorithms have been implemented in the model checkers MCHyper and bounded model checker HyperQube. As discussed in this paper, HyperLTL is limited to synchronous hyperproperties.

FOL[E] can express a limited form of asynchronous hyperproperties. As shown in [9], FOL[E] is subsumed by HyperLTL with additional quantification over predicates. Using such predicates as "markers," one can relate different positions in different traces. However, only a finite number of such predicates is available in each formula. S1S[E] is known to be strictly more expressive than FOL[E] [9], and conjectured to subsume H_μ [15]. For $S1S[E]$ and H_μ, the model checking problem is in general undecidable; for H_μ, two fragments, the k-synchronous, k-context bounded fragments, have been identified for which model checking remains decidable [15]. Even though some asynchronous properties can be expressed in these decidable fragments of H_μ, there is no systematic study to characterize practical properties that can be encoded. Like S1S[E] and H_μ, asynchronous HyperLTL has an (in general) undecidable model checking problem. However, in this paper we have identified decidable fragments of asynchronous HyperLTL that can express observational determinism, noninterference, and linearizability. A-HLTL is thus the first logic for hyperproperties that can express the major asynchronous hyperproperties of interest within decidable fragments. Furthermore, asynchronous HyperLTL is the first logic for asynchronous hyperproperties with a practical model checking algorithm.

8 Conclusion

We have introduced A-HLTL, a temporal logic to describe asynchronous hyperproperties. This logic extends HyperLTL with *trajectory* modalities, which control when a trace proceeds and when it stutters. Synchronous HyperLTL corresponds to a trajectory that always moves all paths in a lock-step manner. This notion of trajectory allows to define formulas that are invariant under stuttering, paving the way for relevant model-checking optimizations such a partial order reduction and abstraction-refinement techniques in the context of hyperproperties. We show that model-checking A-HLTL formulas is in general undecidable, and identify two fragments of A-HLTL formulas, which cover a rich set of security requirements and can be decided by a reduction to HyperLTL model-checking. This in turn has allowed us to the reuse the existing model-checker MCHyper.

Future work includes the study of larger decidable fragments (that encompass both fragments studied here), extending the logic allowing several trajectory modalities, as well as their implementation in practical tools. Extending bounded model-checking [17] to A-HLTL is another interesting research problem. Asynchronous hyperproperties are important for applying a logic-based verification approach to verify hyperproperties for *software* programs, because the relative speed of the execution of programs depends on many factors like the compiler, hardware, execution platform and concurrent running programs, that the analysis must tolerate. Therefore, future work includes adapting techniques for infinite-state software model-checking, like deductive methods, abstraction, etc. to verify A-HLTL properties of software systems.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**, 181–185 (1985)
2. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88313-5_22
3. Barringer, H., Kuiper, R., Pnueli, A.: A really abstract concurrent model and its temporal logic. In: Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL 1986), pp. 173–183. ACM (1986)
4. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. CoRR, abs/2104.14025 (2021)
5. Bonakdarpour, B., Sanchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018, Part II. LNCS, vol. 11245, pp. 8–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_2
6. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
7. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
8. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
9. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J.: The hierarchy of hyperlogics. In: Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019), pp. 1–13. IEEE (2019)
10. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Dillig, I., Tasiran, S. (eds.) CAV 2019, Part I. LNCS, vol. 11561, pp. 121–139. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_7
11. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3

12. Finkbeiner, B., Rabe, M.N., Sánchez, C.: A temporal logic for hyperproperties. CoRR, abs/1306.6657 (2013)
13. Finkbeiner, B., Zimmermann, M.: The first-order logic of hyperproperties. In: 34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, 8–11 Mar 2017, Hannover, Germany, pp. 30:1–30:14 (2017)
14. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
15. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Automata and fixpoints for asynchronous hyperproperties. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021)
16. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
17. Hsu, T.-H., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: TACAS 2021, Part I. LNCS, vol. 12651, pp. 94–112. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_6
18. Lamport, L.: "Sometime" is sometimes "not never" - on the temporal logic of programs. In: Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages (POPL 1980), pp. 174–185. ACM Press (1980)
19. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer-Verlag, New York (1995). https://doi.org/10.1007/978-1-4612-4222-2
20. Namjoshi, K.S., Tabajara, L.M.: Witnessing secure compilation. In: Beyer, D., Zufferey, D. (eds.) VMCAI 2020. LNCS, vol. 11990, pp. 1–22. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_1
21. Pnueli, A.: The temporal logic of programs. In: Symposium on Foundations of Computer Science (FOCS), pp. 46–57 (1977)
22. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054170
23. Post, E.L.: A variant of a recursively unsolvable problem. Bull. Am. Math. Soc. **52**, 264–268 (1946)
24. Rabe, M.N.: A Temporal Logic Approach to Information-flow Control. PhD thesis, Saarland University (2016)
25. Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. High. Order Symb. Comput. **14**(1), 59–91 (2001)
26. Sipser, M.: Introduction to the Theory of Computation. MIT Press, Boston (2012)
27. Wang, Y., Zarei, M., Bonakdarpour, B., Pajic, M.: Statistical verification of hyperproperties for cyber-physical systems. ACM Trans. Embed. Comput. Syst. (TECS) **18**(5s), 92:1–92:23 (2019)
28. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW), p. 29 (2003)