Verifying Asynchronous Hyperproperties in Reactive Systems

RAVEN BEUTNER, CISPA Helmholtz Center for Information Security, Germany
BERND FINKBEINER, CISPA Helmholtz Center for Information Security, Germany

Hyperproperties are system properties that relate multiple execution traces and commonly occur when specifying information-flow and security policies. Logics like HyperLTL utilize explicit quantification over execution traces to express temporal hyperproperties in reactive systems, i.e., hyperproperties that reason about the temporal behavior along infinite executions. An often unwanted side-effect of such logics is that they compare the quantified traces *synchronously*. This prohibits the logics from expressing properties that compare multiple traces asynchronously, such as Zdancewic and Myers's *observational determinism*, McLean's *non-inference*, or *stuttering refinement*. We study the model-checking problem for a variant of *asynchronous HyperLTL* (A-HLTL), a temporal logic that can express hyperproperties where multiple traces are compared across timesteps. In addition to quantifying over system traces, A-HLTL features secondary quantification over stutterings of these traces. Consequently, A-HLTL allows for a succinct specification of many widely used asynchronous hyperproperties. Model-checking A-HLTL requires finding suitable stutterings, which, thus far, has been only possible for very restricted fragments or *terminating* systems. In this paper, we propose a novel game-based approach for the verification of arbitrary $\forall^*\exists^*$ A-HLTL formulas in *reactive* systems. In our method, we consider the verification as a game played between a verifier and a refuter, who challenge each other by controlling parts of the underlying traces and stutterings. A winning strategy for the verifier then corresponds to concrete witnesses for existentially quantified traces and asynchronous alignments for existentially quantified stutterings. We identify fragments for which our game-based interpretation is complete and thus constitutes a finite-state decision procedure. We contribute a prototype implementation for finite-state systems and report on encouraging experimental results.

CCS Concepts: • **Theory of computation → Modal and temporal logics**; **Logic and verification**; **Verification by model checking**; • **Security and privacy → Logic and verification**.

Additional Key Words and Phrases: Temporal Logics, HyperLTL, Asynchronous HyperLTL, Model-Checking, Game-based Semantics, Observational Determinism, Refinement, Hyperliveness

ACM Reference Format:
Raven Beutner and Bernd Finkbeiner. 2025. Verifying Asynchronous Hyperproperties in Reactive Systems. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 352 (October 2025), 27 pages. https://doi.org/10.1145/3763130

1 Introduction

In 2008, Clarkson and Schneider [2008] coined the term *hyperproperties* for the rich class of system requirements that relate multiple computations. Contrary to traditional trace properties (i.e., properties that reason about individual executions, formally defined as *sets of traces*), hyperproperties (formally defined as *sets of sets of traces*) capture the interaction of multiple computations. This covers a wide range of requirements, including information-flow policies [Goguen and Meseguer 1982; Guarnieri et al. 2020], robustness [Biewer et al. 2022], continuity [Chaudhuri et al. 2012], knowledge [Beutner et al. 2023; Bozzelli et al. 2015], and linearizability [Herlihy and Wing 1990]. For example, Zdancewic and Myers [2003]'s seminal definition of *observational determinism* (OD)

Authors' Contact Information: Raven Beutner, raven.beutner@cispa.de, CISPA Helmholtz Center for Information Security, Germany; Bernd Finkbeiner, finkbeiner@cispa.de, CISPA Helmholtz Center for Information Security, Germany.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART352
https://doi.org/10.1145/3763130

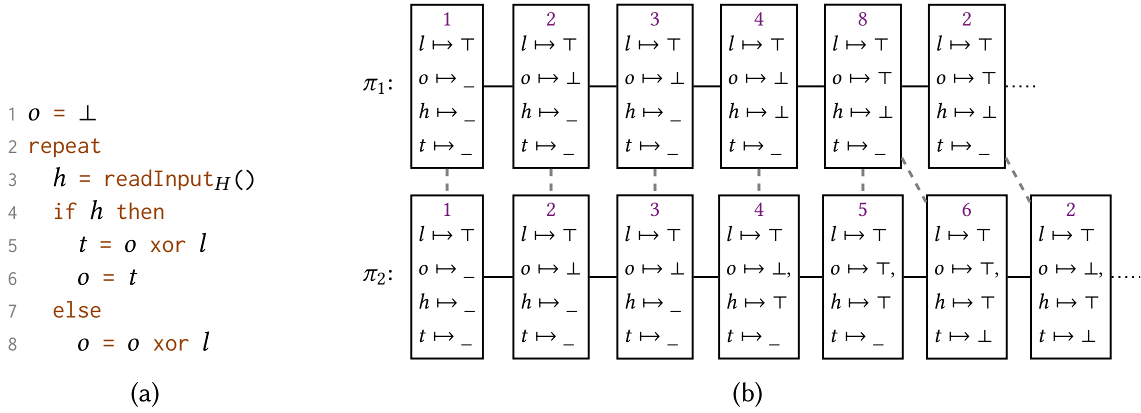

Fig. 1. Figure 1a depicts a Boolean program over variables o, l, h, and t. In Figure 1b, we depict two executions π_1, π_2. Each state contains the current program line and the current value of all variables (initially, we set the low-security input l to $\top$). On π_1, the read in line 3 sets h to $\bot$, and on π_2 it assigns h to $\top$.

requires that all *pairs* of executions with the same low-security input have the same sequence of low-security observations. Likewise, McLean [1994]'s *non-inference* requires that for every execution of the system, there exists a matching execution that has the same (sequence of) low-security observations despite having a fixed "dummy" high-security input; both are hyperproperties.

Missing from Clarkson and Schneider [2008]'s general definition was, however, a concrete specification language for hyperproperties. The introduction of HyperLTL [Clarkson et al. 2014] sparked an extensive development of *temporal* logics for expressing and reasoning about hyperproperties in reactive systems, i.e., systems that continuously interact with an environment and thus produce *infinite* execution traces. HyperLTL extends linear-time temporal logic (LTL) [Pnueli 1977] with explicit quantification over traces in a system. For example, we can express a simplified form of observational determinism as $\varphi_{OD}^{syn} := \forall \pi_1. \forall \pi_2. (l_{\pi_1} = l_{\pi_2}) \rightarrow \square(o_{\pi_1} = o_{\pi_2})$, stating that all pairs of executions π_1, π_2 with initially the same low-security input (modeled via variable l), should globally (expressed using LTL's globally operator $\square$) produce the same output (modeled by o).

Crucially, the semantics of HyperLTL – and that of most other logics for temporal hyperproperties [Ábrahám et al. 2020; Ábrahám and Bonakdarpour 2018; Bajwa et al. 2023; Beutner and Finkbeiner 2023b; Beutner et al. 2023; Coenen et al. 2019a; Dimitrova et al. 2020; Finkbeiner et al. 2020; Giacomo et al. 2021; Gutsfeld et al. 2020; Rabe 2016] – is *synchronous*. That is, the logic can relate multiple traces in a system, but – during the evaluation of the LTL body – time progresses in lock-step on all traces. This is too restrictive for many properties. As a simple example, consider the program in Figure 1a. The program branches on the (Boolean) high-security input h and updates o in either branch to o `xor` l. Initially equal values of l thus imply that the sequence of outputs is the same; the program satisfies Zdancewic and Myers [2003]'s original definition of OD. The program does, however, *violate* the synchronous HyperLTL property expressed in φ_{OD}^{syn}; the update of o is delayed by one step in the first branch (due to the intermediate write to t in line 5), so the LTL body is violated during its synchronous evaluation. Figure 1b depicts two example executions of Figure 1a, illustrating how different executions delay the time point where the output changes.

Asynchronous Hyperproperties. Many security and information-flow properties – particularly in the study of *distributed* programs – thus cannot be expressed in HyperLTL's rigid synchronous semantics. This points to a sharp contrast: Hyperproperties, per Clarkson and Schneider [2008]'s definition, were never "synchronous" and, instead, loosely defined as *sets of sets of traces* and prominent properties [McCullough 1988; McLean 1994; Zdancewic and Myers 2003] only reason

about sequences of events, without any form of synchronous timesteps. Yet, most existing *logics* for expressing temporal hyperproperties enforce a synchronous traversal of all traces.

Asynchronous HyperLTL. It turns out that we can significantly extend the expressiveness of HyperLTL – allowing us to precisely capture many security properties out of reach for synchronous hyperlogics – using a simple idea: *stuttering*. Formally, Asynchronous HyperLTL (A-HLTL for short) extends HyperLTL with explicit quantification over stutterings of system traces [Baumeister et al. 2021]. In this paper, we use a novel (yet equivalent) variant of A-HLTL that directly quantifies over stuttered traces instead of using the so-called trajectories used by Baumeister et al. [2021] (see Section 4 for details). Given a trace π in the system, we write $\beta \blacktriangleright \pi$ to denote that the trace β is a fair stuttering of trace π. In A-HLTL, we can then state

$$\forall \pi_1.\forall \pi_2.\exists \beta_1 \blacktriangleright \pi_1.\exists \beta_2 \blacktriangleright \pi_2.\, (l_{\beta_1} = l_{\beta_2}) \rightarrow \Box(o_{\beta_1} = o_{\beta_2}) \qquad (\varphi_{OD})$$

requiring that for *every* pair of traces π_1, π_2 in the system, there *exist* stutterings β_1 of π_1 and β_2 of π_2, such that the stuttered traces β_1, β_2 satisfy $(l_{\beta_1} = l_{\beta_2}) \rightarrow \Box(o_{\beta_1} = o_{\beta_2})$. φ_{OD} thus requires that all pairs of traces π_1, π_2 with an initially equal value of l, are *stutter-equivalent* on o, *precisely* capturing Zdancewic and Myers [2003]'s original definition of observational determinism. The program in Figure 1a satisfies φ_{OD}: any two traces traverse the same sequence of output values and can, therefore, be aligned by stuttering appropriately. In Figure 1b, we visualize a possible stuttering that aligns the outputs using the dashed lines.

Verifying Asynchronous HyperLTL. Baumeister et al. [2021] and Hsu et al. [2023] demonstrate that A-HLTL is an expressive logic that allows for succinct high-level specifications of, e.g., many commonly used information-flow policies and asynchronous *refinement* properties (see Section 8 for more details). In particular, we can use A-HLTL to directly *quantify* over stutterings without manually identifying alignment points (such as positions where the output of a system changes, or non-visible steps in a refinement property); see Section 8 for an extensive discussion. While A-HLTL seems like a simple extension of HyperLTL, the quantification over stutterings significantly complicates model-checking, so we cannot apply existing algorithms developed for HyperLTL (which often heavily rely on HyperLTL's synchronous evaluation by, e.g., using automata to summarize trace combinations [Beutner and Finkbeiner 2023a; Finkbeiner et al. 2015]). Instead, existing approaches for A-HLTL either employ a bounded unrolling of the system and are thus limited to *terminating* systems (i.e., systems that reach a final state after a *fixed* number of steps) [Hsu et al. 2023] or consider restricted fragments of A-HLTL that can be manually reduced to HyperLTL [Baumeister et al. 2021].

Game-Based Verification for A-HLTL. In this paper, we propose a novel method for the verification of A-HLTL in reactive systems that can verify relevant properties well beyond the fragments supported by previous methods. In our approach, we interpret the verification of an A-HLTL formula as a game between a verifier and a refuter; similar to successful approximations for synchronous HyperLTL [Beutner and Finkbeiner 2022a; Coenen et al. 2019b]. In our game, we construct the traces and stutterings *step-wise* and yield the control of traces – *and their stutterings* – to the players. Intuitively, in each step, the refuter can extend all universally quantified traces and decide on whether universally quantified stutterings should be progressed or stuttered. The verifier can make similar decisions for all existentially quantified traces and stutterings. We provide an overview in Section 2. We show that our game-based approximation is sound: If the verifier wins the game, the A-HLTL formula is satisfied. As model-checking of A-HLTL is, in general, undecidable (even in finite-state systems!) [Baumeister et al. 2021], our approach is necessarily incomplete. We identify fragments of A-HLTL for which our game-based interpretation is complete and thus

constitutes a finite-state *decision procedure*. In particular, we prove that our method is complete for the fragments supported by previous approaches [Baumeister et al. 2021; Hsu et al. 2023].

Implementation. Our game-based verification method applies to finite and infinite-state systems (see Section 8). To compare with existing approaches and tools, we implement our verification approach for finite-state transition systems in a prototype tool and compare it against a bounded model-checking approach [Hsu et al. 2023].

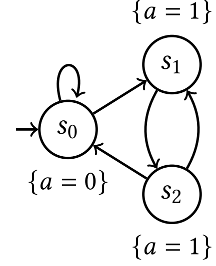

Fig. 2. Example system

Structure and Contributions. In short, our contributions include the following: **(1)** We introduce a simpler (yet equivalent) variant of A-HLTL that directly quantifies over stuttering (Section 4); **(2)** We propose a novel game-based semantics for the verification of asynchronous hyperproperties specified in A-HLTL (Section 5); **(3)** We identify fragments for which our game-based interpretation is complete and thus yields a decision procedure for finite-state systems. Most notably, this includes all admissible formulas [Baumeister et al. 2021], for which we prove that the verifier can follow a canonical ("maximal") stuttering strategy (Section 6); **(4)** We implement our approach for finite-state systems and evaluate it on existing benchmarks (Section 7).

2 Overview

In this section, we illustrate how we can use games to verify hyperproperties that quantify over stutterings. Consider the 3-state transition system $\mathcal{T}$ in Figure 2, where each state assigns variable a to some integer value, and the following A-HLTL formula

$$\forall \pi_1.\exists \pi_2.\exists \beta_1 \blacktriangleright \pi_1.\exists \beta_2 \blacktriangleright \pi_2. \bigcirc \square (a_{\beta_1} \neq a_{\beta_2}). \qquad (\varphi_{fair})$$

This formula requires that for any execution π_1, there exists some execution π_2 and some stutterings β_1, β_2 (of π_1 and π_2, respectively) such that, starting from the second step (using LTL's next operator $\bigcirc$), the value of a differs between β_1 and β_2.[1] This property holds on $\mathcal{T}$. For example, take the trace $\pi_1 = \{a = 0\}\{a = 0\}\big(\{a = 1\}\big)^\omega$. We can match this execution by choosing $\pi_2 = \{a = 0\}\{a = 1\}\{a = 1\}\big(\{a = 0\}\big)^\omega$, and align both traces by picking stutterings $\beta_1 = \{a = 0\}\{a = 0\}\{a = 0\}\big(\{a = 1\}\big)^\omega$, and $\beta_2 = \pi_2$ (note that the "obvious" choice $\pi_2 = \{a = 0\}\{a = 1\}\big(\{a = 0\}\big)^\omega$ is *not* a trace in $\mathcal{T}$).

When verifying this property directly, we would need to, for each choice of π_1 (where quantification ranges over infinite traces in $\mathcal{T}$), provide a witnessing trace π_2 *and* find an appropriate stuttering for each combination of π_1, π_2. Instead of tackling this problem directly, we *approximate* the verification as a game between a verifier and a refuter.[2] The main idea of our game is to view paths for π_1 and π_2 and the stutterings β_1, β_2 as the outcome of an infinite-duration two-player game between a refuter (who controls *universally* quantified traces and stutterings) and a verifier (who controls *existentially* quantified traces and stutterings). For our example formula φ_{fair}, the refuter constructs the (universally quantified) trace π_1 step-wise by moving along the transitions of $\mathcal{T}$. The verifier can react by moving through a separate copy of $\mathcal{T}$, thereby generating a trace for π_2, and controls the stutterings β_1, β_2. Each infinite game-play thus generates infinite traces for

[1]The A-HLTL formula we consider here is meant as a simple example that demonstrates the key ideas underlying our verification method. Still, if we interpret $a = v$ (for $v \in \{0, 1\}$) as "a grant was given to process v", we can view φ_{fair} as a simplified *fairness* property: it requires that for each execution, some other execution results in the exact opposite grant assignment (up to stuttering), i.e., both processes can receive the grant *equally often*.

[2]Such approximations have proven successful in the verification of synchronous HyperLTL [Beutner and Finkbeiner 2022a, 2024, 2025c; Coenen et al. 2019b], where finite-state model-checking is decidable, and games offer a computationally cheaper (incomplete) approximation. In our setting of A-HLTL, model-checking is *undecidable*, and we design a novel game as a sound *approximation* of A-HLTL.

π_1 and π_2 and stutterings β_1, β_2 of π_1, π_2, respectively; the objective of the verifier is then to ensure that the stutterings β_1, β_2 (together) satisfy $\bigcirc\Box(a_{\beta_1} \neq a_{\beta_2})$, the LTL body of φ_{fair}.

Graph-based Games. To formalize the interaction of refuter and verifier, we construct a graph-based game. The game consists of a set of vertices connected via edges, where each vertex is assigned to either the refuter or the verifier. The game progresses by moving a single token from vertex to vertex, and the player controlling the current vertex can decide which outgoing edge the token should take. The outgoing edges of a given vertex thus determine the possible decision a player can make in the given situation (we give more details on graph-based games in Section 3).

Verification Game. There are two principled ideas underlying our verification game for A-HLTL: Windows of states, and relative pointers. The refuter (resp. verifier) controls trace π_1 (resp. π_2) by moving through $\mathcal{T}$. To accommodate the fact that β_1 and β_2 are stutterings of π_1 and π_2 (i.e., in the ith step of stuttering β_1 can equal the jth step of π_1 for any $j \leq i$, depending on the speed of the stuttering), we do not maintain a single state but a *window* of states, which we can think of a *sliding window* of a trace. We then identify the stutterings β_1, β_2 as *relative pointers* into the state-windows of π_1 and π_2. Intuitively, this pointer identifies the current position of the stuttering, i.e., β_1 (resp. β_2) points to one position of (the state-window of) π_1 (resp. π_2). Whenever we require the current state of β_1, i.e., we fetch the state in π_1's window that is pointed to by the β_1 pointer.

Each vertex then belongs to one of three game states: the Ⓤ-stage (update-stage), the Ⓐ-stage, or Ⓔ-stage. The verifier controls all vertices in the Ⓔ-stage, and the refuter those in the Ⓐ-stage (vertices in the Ⓤ-stage have a unique successor so it does not matter which player controls them). Whenever the game is in the Ⓤ-stage, we consider the current states of β_1, β_2, and evaluate one step of the LTL formula $\bigcirc\Box(a_{\beta_1} \neq a_{\beta_2})$. In order to win, the verifier thus needs to make sure that, whenever the game is in the Ⓤ-stage (except on the first visit), $a_{\beta_1} \neq a_{\beta_2}$ holds.[3] After the Ⓤ-stage, we progress to the Ⓐ-stage. In the Ⓐ-stage, the refuter can progress the (universally quantified) π_1; the state window for π_2 remains unchanged. As we track a window of states, this corresponds to *appending* a state to that window along $\mathcal{T}$'s transitions. In the Ⓔ-stage, the verifier can respond by appending a state to π_2's window. Additionally, the verifier can decide if the (*existentially quantified*) stutterings β_1 and β_2 should progress to the next state in the state window of π_1 and π_2, respectively. The verifier can thus (implicitly) control the stutterings β_1, β_2 by controlling the relative pointers. By advancing the β_1-pointer on π_1 (i.e., move the β_1-pointer to the next state in the state window assigned to π_1), the stuttering β_1 does a proper (non-stuttering) step; if the verifier leaves the β_1-pointer unchanged, it effectively performs a stuttering step by repeating the same state it pointed to in the previous round.

Winning Strategy. In our example, the system $\mathcal{T}$ consists of finitely many states, so the resulting graph-based game contains only finitely many vertices. Figure 3 depicts a fragment of a winning strategy for the verifier.[4] Each vertex maps π_1 and π_2 to a non-empty window (sequence) of states, the stutterings β_1, β_2 are represented as arrows. Our game begins in initial vertex v_0, where π_1 and π_2 map to the singleton window $[s_0]$ containing $\mathcal{T}$'s initial state, and the β_1 and β_2 pointers point to the unique first position. In v_1 (in the Ⓐ-stage), the refuter can now choose a successor state for π_1 by either appending state s_0 (i.e., move to v_2) or state s_1 (i.e., move to v_8) to π_1's state window. We focus on the former case. In v_2 (which is in the Ⓔ-stage), the verifier can append a state

[3] To track this temporal property, we translate the LTL formula to a deterministic automaton and track the state of this automaton within each Ⓤ-stage vertex. For simplicity, we omit this automaton in this overview section.

[4] A (positional) strategy for the verifier is a function that, for each vertex controlled by the verifier, fixes a concrete successor vertex (see Section 3). We visualize a strategy as a restriction of the game graph. That is, we include all possible successor vertices for all refuter-controlled vertices, but fix one particular outgoing edge for all verifier-controlled vertices.

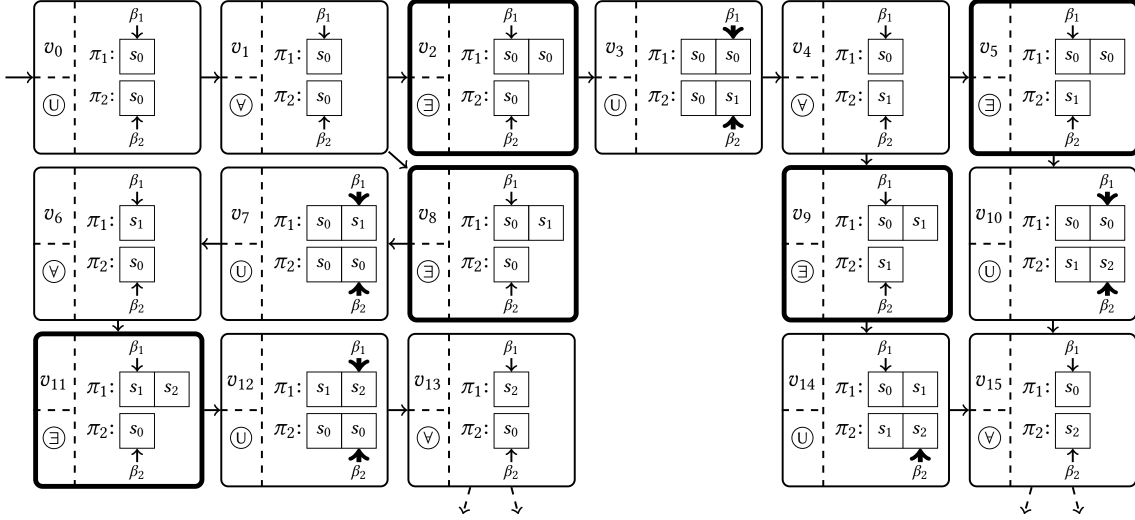

Fig. 3. We depict (parts of) a winning strategy for the verifier in the verification game for the system in Figure 2. Each vertex maps π_1, π_2 to some window of states and tracks the current stage in {Ⓤ, Ⓐ, ∃⃝}. We label the vertices $v_0, \cdots, v_{15}$. The verifier controls all vertices in the ∃⃝-stage, marked by a thick border.

to π_2's window *and* decide if the β_1 and β_2 pointers should be progressed. The winning strategy depicted here moves to v_3, i.e., appends state s_1 and advances both β_1 and β_2. In v_3 (which is in the Ⓤ-stage), β_1 points to state s_0, and β_2 points to s_1, which differ in the evaluation of a, as required by $\bigcirc\Box(a_{\beta_1} \neq a_{\beta_2})$. From v_3, we (deterministically) move to v_4 by removing the first position of the state windows for π_1 and π_2: The β_1 and β_2 pointers (which give the current position of the stuttering) point to the second position in each state window, so we can safely drop all positions preceding the current position of the stuttering. In v_4, the refuter can again append state s_0 (i.e., move to v_5) or s_1 (i.e., move to v_9) to π_1's state window. We focus on the latter case. In v_9, the current window for π_2 ends in state s_1, so the verifier has no choice but to append s_2 (as it has to respect $\mathcal{T}$'s transitions). A possible move for the verifier *would* be to progress *both* the β_1 and β_2 pointers, so they would point to states s_1 and s_2, respectively. However, this would lose the game for the verifier as, in the next Ⓤ-stage, $a_{\beta_1} \neq a_{\beta_2}$ is violated. Instead, the verifier moves to vertex v_{14}, i.e., it only progresses the β_2 pointer, and leaves the β_1-pointer unchanged, effectively stuttering β_1 (i.e., the β_1 pointer still points to state s_0, repeating the same state as in the previous round). In v_{14}, the stutterings β_1 and β_2 thus point to states s_0 and s_2, respectively, satisfying $a_{\beta_1} \neq a_{\beta_2}$. In v_{14}, we, again, (deterministically) remove the first position of the π_2 window. Additionally, we can shorten π_1's window by removing all positions trailing the β_1-pointer, thereby ensuring that the state windows never grow to more than 2 states (i.e., we deterministically move to vertex v_{15}).

The initial sketch in Figure 3 can be extended to a full strategy which **(1)** ensures that $\bigcirc\Box(a_{\beta_1} \neq a_{\beta_2})$ holds (where we evaluate one step whenever in the Ⓤ-stage), and **(2)** advances the β_1 and β_2 pointers infinitely often (i.e., stutterings β_1 and β_2 are fair). Notably, our game is designed such that it underapproximates the power of existential quantification: If the verifier can construct appropriate witnesses for π_2, β_1, β_2, no matter how the refuter constructs on π_1, then we can also find witnesses for π_2, β_1, β_2 when given the entire (infinite) trace π_1 (as in the semantics of A-HLTL); using the game we thus proved that $\mathcal{T}$ satisfies φ_{fair}.

We emphasize that, even though this example is deliberately simple (for demonstration purposes), our method is the first that can verify φ_{fair} automatically: The system in Figure 2 is clearly not terminating [Hsu et al. 2023], and φ_{fair} is not admissible (cf. Section 6.3) [Baumeister et al. 2021].

3 Preliminaries

We write $\mathbb{N}$ for the set of natural numbers and $\mathbb{B} := \{\top, \bot\}$ for the set of Booleans. Given a set X, we write X^ω for the set of infinite sequences over X and $X^{\leq n}$ for the set of finite sequences of length at most $n \in \mathbb{N}$. For $u \in X^{\leq n}$, $i \in \mathbb{N}$, and $x \in X$, we define: $|u|$ as the length of u, $u(i) \in X$ as the ith element in u (starting with the 0th), and $u \cdot x$ as the sequence where we append x to u. For $i, j \in \mathbb{N}$, we define $u[i, j] := u(i)u(i+1) \cdots u(j)$ if $|u| > j$, and otherwise (if $|u| \leq j$) $u[i, j] := u(i)u(i+1) \cdots u(|u|-1)$. Finally, we define $u[i, j) := u[i, j-1]$. We denote unnamed functions using λ notation: For example, $\lambda n.\, n + 1$ denotes the function $\mathbb{N} \to \mathbb{N}$ that maps each $n \in \mathbb{N}$ to $n + 1$.

First Order Theories. HyperLTL [Clarkson et al. 2014] and A-HLTL [Baumeister et al. 2021; Hsu et al. 2023] are evaluated over transition systems and access the current state using finitely many *atomic propositions* (APs), which we can think of as Boolean variables. However, in many cases, the variables of a system are not Boolean but come from richer domains (e.g., the set of integers). We extend A-HLTL by including first-order formulas (modulo some fixed first-order theory) as atomic expressions. That is, instead of accessing traces at the level of atomic propositions, we can reason about relational formulas over complex data types and use interpreted predicates like $=, \neq, \leq, \dots$ [Beutner and Finkbeiner 2025a]. Formally, we assume some first-order signature and some fixed background theory $\mathfrak{T}$ with universe $\mathbb{V}$ (which we can think of as the set of all values). For a set of variables $\mathcal{X}$, we write $\mathcal{F}_\mathcal{X}$ for the set of all first-order formulas over variables in $\mathcal{X}$. Given a formula $\theta \in \mathcal{F}_\mathcal{X}$ and a variable assignment $A : \mathcal{X} \to \mathbb{V}$, we write $A \models^\mathfrak{T} \theta$ if θ is satisfied by A (modulo $\mathfrak{T}$) [Barwise 1977].

Transition Systems. As the basic computation model, we use state-based transition systems. For this, let $\mathcal{X}$ be a fixed finite set of system variables. A *transition system* (TS) is a tuple $\mathcal{T} = (S, s_0, \kappa, \ell)$, where S is a (possibly infinite) set of states, $s_0 \in S$ is an initial state, $\kappa : S \to (2^S \setminus \{\emptyset\})$ is a transition function, and $\ell : S \to (\mathcal{X} \to \mathbb{V})$ is a function that maps each state to a variable assignment over $\mathcal{X}$. A path in $\mathcal{T}$ is an infinite sequence $p = s_0 s_1 s_2 \cdots \in S^\omega$ (starting in s_0) such that $s_{i+1} \in \kappa(s_i)$ for all $i \in \mathbb{N}$. We write $Paths(\mathcal{T}) \subseteq S^\omega$ for the set of all paths in $\mathcal{T}$. Given a path $p = s_0 s_1 s_2 \cdots$, the associated trace is given by applying ℓ pointwise, i.e., $\ell(p) := \ell(s_0)\ell(s_1)\ell(s_2) \cdots \in (\mathcal{X} \to \mathbb{V})^\omega$. We write $Traces(\mathcal{T}) \subseteq (\mathcal{X} \to \mathbb{V})^\omega$ for the set of all traces generated by $\mathcal{T}$. Given two trace $t, t' \in (\mathcal{X} \to \mathbb{V})^\omega$, we say t' is a *fair stuttering of* t (written $t' \blacktriangleright t$) if t' is obtained from t by stuttering any position for an arbitrary (but *finite*) number of steps. Formally, $t' \blacktriangleright t$ iff there exists a function $f : \mathbb{N} \to \mathbb{N}$ such that **(1)** $\forall i \in \mathbb{N}.\, t'(i) = t\big(f(i)\big)$, **(2)** f is monotonically increasing (i.e., $i \geq j$ implies $f(i) \geq f(j)$), and **(3)** f is surjective (i.e., $\forall i \in \mathbb{N}.\, \exists j \in \mathbb{N}.\, f(j) = i$).

Büchi Automata. During the game, we need to track if the traces and stutterings constructed by the verifier satisfy the LTL body of the A-HLTL formula. To accomplish this, we translate the LTL formula to a *deterministic Büchi automaton* (DBA). A DBA is similar to a DFA over finite words but accepts *infinite* words. We can thus run this automaton in parallel with the (infinite) behavior of refuter and verifier, and determine if their infinite play satisfies the LTL formula.[5] A (DBA) over alphabet Σ is a tuple $\mathcal{A} = (Q, q_0, \delta, F)$, where Q is a finite set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \to Q$ is a deterministic transition function, and $F \subseteq Q$ is a set of accepting states. Given a word $u \in \Sigma^\omega$, there exists a unique run $q_0 q_1 \cdots \in Q^\omega$ of $\mathcal{A}$ (starting in q_0) defined by $q_{i+1} = \delta(q_i, u(i))$ for all $i \in \mathbb{N}$. The run is accepting if it visits states in F *infinitely often*. We write $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ for the set of words where the unique run of $\mathcal{A}$ is accepting.

[5]Note that not every LTL formula can be translated to a deterministic Büchi automaton. Instead, we can use slightly more complex deterministic *parity* automata, which capture every ω-regular property [Piterman 2007]. Throughout this paper, we nevertheless use simpler Büchi automata to simplify the presentation.

Büchi Games. As a concrete game formalism, we use graph-based games, where two players move a token from vertex to vertex along the edges of the graph. Our later game will be designed such that the outgoing edges in the graph precisely capture the ability of either player in the current game situation (e.g., append a state to a state window, progress a stuttering, etc.). The objective of the verifier is to ensure that the infinite game play satisfies the LTL body of the A-HLTL formula. As we track this LTL body using a Büchi automaton, we naturally obtain a Büchi winning objective for the verifier. Formally, a *Büchi game* is a tuple $\mathcal{G} = (V_{\mathfrak{V}}, V_{\mathfrak{R}}, \alpha, F)$, where $V := V_{\mathfrak{V}} \uplus V_{\mathfrak{R}}$ is a (possibly infinite) set of vertices (vertices in $V_{\mathfrak{V}}$ are controlled by the verifier, those in $V_{\mathfrak{R}}$ by the refuter), $\alpha : V \to (2^V \setminus \{\emptyset\})$ is a transition function, and $F \subseteq V$ is a set of accepting vertices. A play in $\mathcal{G}$ is an infinite sequence $\rho \in V^\omega$ such that $\rho(i+1) \in \alpha(\rho(i))$ for all $i \in \mathbb{N}$. The play ρ is won by $\mathfrak{V}$ if it visits states in F infinitely often. A positional strategy (for the verifier) (cf. [Martin 1975]) is a function $\sigma : V_{\mathfrak{V}} \to V$ such that $\sigma(v) \in \alpha(v)$ for all $v \in V_{\mathfrak{V}}$. A play $\rho \in V^\omega$ is compatible with σ if for every $i \in \mathbb{N}$ with $\rho(i) \in V_{\mathfrak{V}}$, we have that $\rho(i+1) = \sigma(\rho(i))$. As we have done in Section 2, we can think of a strategy as a restriction of the game's state space: we include all outgoing edges for all vertices controlled by the refuter; for each verifier-controlled vertex $v \in V_{\mathfrak{V}}$ we only include the edge to vertex $\sigma(v)$. The verifier wins vertex $v \in V$ if there exists a strategy σ such that all σ-compatible plays ρ (i.e., all infinite paths in the restricted vertex space) that start in v (i.e., $\rho(0) = v$) are won by the verifier. Given a set of vertices $X \subseteq V$, we write $wins_{\mathcal{G}}(\mathfrak{V}, X)$ if the verifier wins $\mathcal{G}$ from any vertex $v \in X$.

4 Asynchronous HyperLTL

HyperLTL extends LTL [Pnueli 1977] with explicit quantification over traces [Clarkson et al. 2014]. A-HLTL then extends the primary quantification over traces with secondary quantification over stutterings of these traces.

Remark 1. *In the original variant of A-HLTL [Baumeister et al. 2021; Hsu et al. 2023], quantification over stutterings was achieved by quantifying over so-called* trajectories. *For example, $\forall \pi_1.\forall \pi_2.\mathsf{E}\tau.\square(o_{\pi_1,\tau} = o_{\pi_2,\tau})$ states that for all pairs of traces π_1, π_2, there exists some trajectory τ that aligns both traces on the output o. Here, the pairs (π_1, τ) and (π_2, τ) denote* independent *stutterings of π_1 and π_2, respectively. This detour via trajectories leads to a convoluted semantics, as each trajectory* implicitly *quantifies over stutterings for* all *traces, see [Baumeister et al. 2021; Hsu et al. 2023] for details. In this paper, we propose a novel variant of A-HLTL by* explicitly *quantifying over stutterings of traces. Our variant is easier to understand, allows for a simpler semantics, and is* equivalent *to the original variant (see the full version). The above example can be expressed in our new A-HLTL variant as $\forall \pi_1.\forall \pi_2.\exists \beta_1 \blacktriangleright \pi_1.\exists \beta_2 \blacktriangleright \pi_2.\square(o_{\beta_1} = o_{\beta_2})$, i.e., for each trace-trajectory pair (π_1, τ), (π_2, τ), we* explicitly *quantify over a stuttering of that trace.*

Syntax. Let $\mathcal{V} = \{\pi_1, \pi_2, \dots, \pi_n\}$ be a set of *trace variables* and $\mathcal{B} = \{\beta_1, \beta_2, \dots, \beta_k\}$ be a set of *stuttering variables*. We define $\mathcal{X}_{\mathcal{B}} := \{x_\beta \mid x \in \mathcal{X}, \beta \in \mathcal{B}\}$ as the set of system variables indexed by stuttering variables. An A-HLTL formula φ is generated by the following grammar

$$\varphi := \exists \pi.\, \varphi \mid \forall \pi.\, \varphi \mid \phi$$
$$\phi := \exists \beta \blacktriangleright \pi.\, \phi \mid \forall \beta \blacktriangleright \pi.\, \phi \mid \psi$$
$$\psi := \theta \mid \neg \psi \mid \psi \land \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi$$

where $\pi \in \mathcal{V}$ is a trace variable, $\beta \in \mathcal{B}$ is a stuttering variable, and $\theta \in \mathcal{F}_{\mathcal{X}_{\mathcal{B}}}$ is a first-order formula over stuttering-variable-indexed system variables (i.e., $\mathcal{X}_{\mathcal{B}}$). We write $\diamondsuit \psi$ and $\square \psi$ for LTL's derived *eventually* and *globally* operator, respectively.

Semantics. An A-HLTL formula from the above grammar has the form

$$\varphi = \mathbb{Q}\pi_1 \ldots \mathbb{Q}\pi_n.\ \mathbb{Q}\beta_1 \blacktriangleright \pi_{l_1} \ldots \mathbb{Q}\beta_k \blacktriangleright \pi_{l_l}.\ \psi$$

where $\pi_1, \ldots, \pi_n \in \mathcal{V}$ are trace variables, $\beta_1, \ldots, \beta_k \in \mathcal{B}$ are stuttering variables quantifying over stutterungs of traces $\pi_{l_1}, \ldots, \pi_{l_l}$, respectively (where $l_1, \ldots, l_k \in \{1, \ldots, n\}$), $\mathbb{Q} \in \{\forall, \exists\}$ are quantifiers, and ψ is an LTL formula over atoms from $\mathcal{F}_{\mathcal{X}_\mathcal{B}}$ (i.e., first-order formulas over variables from $\mathcal{X}_\mathcal{B}$). Similar to HyperLTL, we first quantify over traces $\pi_1, \ldots, \pi_n$ in our transition system $\mathcal{T}$ in typical first-order semantics. After the n traces are fixed, we proceed to the secondary quantification over stutterings of these traces, i.e., $\mathbb{Q}\beta \blacktriangleright \pi$ quantifies over a fair stuttering of (the previously quantified) trace π. After resolving the quantifier prefix, we are thus left with k stutterings $\beta_1, \ldots, \beta_k$, and can evaluate the LTL formula ψ. Here, each atom in the LTL formula ψ is a first-order formula over variables from $\mathcal{X}_\mathcal{B}$ (modulo $\mathfrak{T}$), i.e., over the system variables indexed by stuttering variables. In such a formula, each variable $x_\beta \in \mathcal{X}_\mathcal{B}$ refers to the current value of x on stuttering β. For example, in φ_{OD}, we used the atomic formula $o_{\beta_1} = o_{\beta_2}$, where "=" is an interpreted predicate symbol from theory $\mathfrak{T}$ and $o_{\beta_1}, o_{\beta_2} \in \mathcal{X}_{\{\beta_1,\beta_2\}}$ are indexed system variables.

To define the semantics formally, we maintain a *trace assignment* $\Pi : \mathcal{V} \rightharpoonup (\mathcal{X} \to \mathbb{V})^\omega$ mapping trace variables to traces (used to evaluate trace quantification), and a *stuttering assignment* $\Delta : \mathcal{B} \rightharpoonup (\mathcal{X} \to \mathbb{V})^\omega$ mapping stuttering variables to traces (used to evaluate stuttering quantification). Given a stuttering assignment Δ and position $i \in \mathbb{N}$, we write $\Delta_{(i)} : \mathcal{X}_\mathcal{B} \to \mathbb{V}$ for the variable assignment to $\mathcal{X}_\mathcal{B}$ in the ith step, defined as $\Delta_{(i)}(x_\beta) := \Delta(\beta)(i)(x)$, i.e., the value of x_β is defined as the value of x in the ith step on stuttering $\Delta(\beta)$. Given a set of traces $\mathbb{T} \subseteq (\mathcal{X} \to \mathbb{V})^\omega$, and position $i \in \mathbb{N}$, we define the semantics of A-HLTL as follows

$$
\begin{array}{lll}
\Pi, \Delta, i \models_\mathbb{T} \exists \pi.\, \varphi & \text{iff} & \exists t \in \mathbb{T}.\, \Pi[\pi \mapsto t], \Delta, i \models_\mathbb{T} \varphi \\
\Pi, \Delta, i \models_\mathbb{T} \forall \pi.\, \varphi & \text{iff} & \forall t \in \mathbb{T}.\, \Pi[\pi \mapsto t], \Delta, i \models_\mathbb{T} \varphi \\
\Pi, \Delta, i \models_\mathbb{T} \exists \beta \blacktriangleright \pi.\, \phi & \text{iff} & \exists t' \in (\mathcal{X} \to \mathbb{V})^\omega.\, t' \blacktriangleright \Pi(\pi) \land \Pi, \Delta[\beta \mapsto t'], i \models_\mathbb{T} \phi \\
\Pi, \Delta, i \models_\mathbb{T} \forall \beta \blacktriangleright \pi.\, \phi & \text{iff} & \forall t' \in (\mathcal{X} \to \mathbb{V})^\omega.\, t' \blacktriangleright \Pi(\pi) \Rightarrow \Pi, \Delta[\beta \mapsto t'], i \models_\mathbb{T} \phi \\
\Pi, \Delta, i \models_\mathbb{T} \theta & \text{iff} & \Delta_{(i)} \models^\mathfrak{T} \theta \\
\Pi, \Delta, i \models_\mathbb{T} \neg\psi & \text{iff} & \Pi, \Delta, i \not\models_\mathbb{T} \psi \\
\Pi, \Delta, i \models_\mathbb{T} \psi_1 \land \psi_2 & \text{iff} & \Pi, \Delta, i \models_\mathbb{T} \psi_1 \text{ and } \Pi, \Delta, i \models_\mathbb{T} \psi_2 \\
\Pi, \Delta, i \models_\mathbb{T} \bigcirc \psi & \text{iff} & \Pi, \Delta, i + 1 \models_\mathbb{T} \psi \\
\Pi, \Delta, i \models_\mathbb{T} \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \exists j \geq i.\, \Pi, \Delta, j \models_\mathbb{T} \psi_2 \text{ and } \forall i \leq k < j.\, \Pi, \Delta, k \models_\mathbb{T} \psi_1.
\end{array}
$$

We first populate a trace assignment Π by following the trace-quantifier prefix over traces in $\mathbb{T}$ and adding traces to Π. For each quantified stuttering $\mathbb{Q}\beta \blacktriangleright \pi$, we then quantify over a stuttering t' of trace $\Pi(\pi)$ and add it to Δ. Note that t' is an arbitrary trace obtained by stuttering $\Pi(\pi)$; it may not be a trace in $\mathbb{T}$. Finally, we can evaluate the LTL body on the traces in Δ following the usual evaluation of boolean and temporal operators. Here, an atomic formula θ holds in step i if the variable assignment $\Delta_{(i)}$ (which assigns values to the indexed variables in $\mathcal{X}_\mathcal{B}$) satisfies θ (modulo theory $\mathfrak{T}$). A transition system $\mathcal{T}$ satisfies an A-HLTL property φ, written $\mathcal{T} \models \varphi$, if $\emptyset, \emptyset, 0 \models_{Traces(\mathcal{T})} \varphi$, where $\emptyset$ denotes a trace/stuttering assignment with an empty domain.

Example 4.1. In most A-HLTL formulas, we quantify (usually existentially) over *one* stuttering for each trace variable. However, A-HLTL can also quantify over multiple stutterings of the same trace [Hsu et al. 2023]. For example, in φ_{OD} from Section 1, the low-security input is fixed *initially* and never changes during evaluation (as in Zdancewic and Myers [2003]'s original definition). If

the low-security input can change over time, the specification needs to take the infinite *sequence* of low-security inputs into account. We can express such an extension as follows

$$\forall \pi_1.\forall \pi_2.\ \forall \beta_1 \blacktriangleright \pi_1.\forall \beta_2 \blacktriangleright \pi_2.\ \exists \beta_3 \blacktriangleright \pi_1.\exists \beta_4 \blacktriangleright \pi_2.\ \square(l_{\beta_1} = l_{\beta_2}) \rightarrow \square(o_{\beta_3} = o_{\beta_4}). \qquad (\varphi_{NI})$$

This formula requires that there *exist* stutterings β_3, β_4 of π_1, π_2 that align the output o (similar to φ_{OD}), assuming the *universally quantified* stutterings β_1, β_2 can align the low-security input l. Phrased differently, any two traces with stutter-equivalent input should produce stutter-equivalent output. △

Example 4.2. Multiple stutterings are also needed to analyze traces w.r.t. different speeds. We illustrate this by examining the various variants of observational determinism found in the literature. Given a low-security input $l \in \mathcal{X}$ and two low-security outputs $a, b \in \mathcal{X}$, the A-HLTL formula

$$\forall \pi_1.\, \forall \pi_2.\, \exists \beta_1 \blacktriangleright \pi_1.\exists \beta_2 \blacktriangleright \pi_2.\, (l_{\beta_1} = l_{\beta_2}) \rightarrow \square\left(a_{\beta_1} = a_{\beta_2} \land b_{\beta_1} = b_{\beta_2}\right)$$

requires that a and b are, *together*, stutter-equivalent on all pairs of traces (similar to the definitions of Terauchi [2008]; Zdancewic and Myers [2003]). However, especially when considering distributed systems, the individual variables on a trace might stem from different distributed components with independent timing. In this case, we can express

$$\forall \pi_1.\, \forall \pi_2.\, \exists \beta_1 \blacktriangleright \pi_1, \beta_2 \blacktriangleright \pi_2, \beta_3 \blacktriangleright \pi_1, \beta_4 \blacktriangleright \pi_2.\, (l_{\beta_1} = l_{\beta_2}) \rightarrow \square\left(a_{\beta_1} = a_{\beta_2} \land b_{\beta_3} = b_{\beta_4}\right),$$

requiring that variables a and b are *individually* stutter equivalent, i.e., we can choose separate stutterings for both outputs (similar to the definition of Bartocci et al. [2023]; Huisman et al. [2006]). △

Example 4.3. McLean [1994]'s notion of non-inference states that for every trace π_1, there should exist some trace π_2 that has the same *sequence* of low-security events but a fixed dummy high-security input. Crucially, McLean [1994]'s definition only reasons about the (ordered) sequence of low-security events, i.e., it does *not* reason about the absolute (synchronous) timesteps. In A-HLTL, we can directly express non-inference as

$$\forall \pi_1.\, \exists \pi_2.\ \exists \beta_1 \blacktriangleright \pi_1, \beta_2 \blacktriangleright \pi_2.\ \square(\bigwedge_{x \in L} x_{\beta_1} = x_{\beta_2}) \land \square(\bigwedge_{x \in H} x_{\beta_2} = \dagger).$$

That is, for every trace π_1, there exists some trace π_2 such that some stutterings β_1, β_2 can align π_1, π_2 so that the low-security variables ($L \subseteq \mathcal{X}$) are equal up to stuttering (so the *sequence of low-security events is the same*). At the same time, the high-security inputs on π_2 ($H \subseteq \mathcal{X}$) should globally be set to some dummy value (which we denote with †). △

We emphasize that while stuttering in itself is not particularly interesting, it is the key technical gadget that *unifies* many of the widely used information-flow properties in the *same* logic (see also Section 8). That is, by extending HyperLTL with the ability to quantify over stutterings of traces, we can suddenly express a much wider range of information-flow policies. Any verification approach for A-HLTL can, therefore, be applied to all those properties (and variants thereof by, e.g., declaring some additional variable as output), instead of "re-inventing the wheel" for every new information-flow property (or variant thereof).

5 Game-based Verification of A-HLTL

We say an A-HLTL formula is a $\forall^*\exists^*$ *formula* if no universal trace and stuttering quantifier occurs in the scope of an existentially quantified trace or stuttering (see, e.g., φ_{OD} and φ_{NI}). The idea of our game-based verification approach is to approximate the quantifier alternation in an A-HLTL formula as a game between a verifier ($\mathfrak{V}$) and a refuter ($\mathfrak{R}$). The refuter controls universally quantified traces

and stutterings, and the verifier responds to each move to the refuter by updating existentially quantified traces and stutterings (cf. Section 2). As all universal quantifiers precede any existential quantification, the verifier updates existentially quantified traces and stutterings based on a subset of the information available in the A-HLTL semantics: In the semantics of a $\forall^*\exists^*$ formula, we can pick existentially quantified traces and stutterings after *all* universally quantified traces and stutterings are fixed. In our game, the verifier picks them step-wise, thus knowing only a finite *prefix*. All formulas studied by Baumeister et al. [2021] and Hsu et al. [2023] are $\forall^*\exists^*$ formulas.

In the following, we let $\mathcal{T} = (S_\mathcal{T}, s_{0,\mathcal{T}}, \kappa_\mathcal{T}, \ell_\mathcal{T})$ be a fixed system and let

$$\varphi = \mathbb{Q}\pi_1 \ldots \mathbb{Q}\pi_n.\ \mathbb{Q}\beta_1 \blacktriangleright \pi_{l_1} \ldots \mathbb{Q}\beta_k \blacktriangleright \pi_{l_k}.\ \psi$$

be a fixed $\forall^*\exists^*$ A-HLTL formula (where $l_1, \ldots, l_k \in \{1, \ldots, n\}$). We define $\mathcal{V}_\forall \subseteq \{\pi_1, \ldots, \pi_n\}$ (resp. $\mathcal{V}_\exists$) as all universally (resp. existentially) quantified trace variables, and $\mathcal{B}_\forall \subseteq \{\beta_1, \ldots, \beta_k\}$ (resp. $\mathcal{B}_\exists$) as all universally (resp. existentially) quantified stuttering variables. For each stuttering variable $\beta \in \mathcal{B}$, we define $base(\beta) \in \mathcal{V}$ as the unique trace that β is based on (i.e., the unique π, where the quantifier prefix contains $\mathbb{Q}\beta \blacktriangleright \pi$). For example, in φ_{NI}, $base(\beta_1) = base(\beta_3) = \pi_1$ and $base(\beta_2) = base(\beta_4) = \pi_2$.

In this section, we will construct a Büchi game – called $\mathcal{G}_{\mathcal{T},\varphi,Z}$ – which, if won by the verifier, implies that $\mathcal{T} \models \varphi$. As we already discussed, the high-level idea of this game is to let the verifier (resp. refuter) control all existentially (resp. universally) quantified traces and stutterings. To formalize this, we will cast this game as a graph-based game, where each vertex in the game records the current state of the game (e.g., the state windows, the positions of the stutterings, etc.). The transitions of the game are then designed such that the set of possible successor vertices precisely corresponds to all possible moves that the players can make (e.g., append a state to a state window, progress a stuttering, etc.).

5.1 Tracking Acceptance and Fairness

While the game progresses, we need to track if the paths and stutterings constructed by the players satisfy ψ (the LTL body of φ). We accomplish this by translating ψ to a DBA. Additionally, we need to ensure that all stutterings are fair, i.e., progress *infinitely often*. To ensure this, we consider the following modified LTL formula

$$\psi_{mod} := \Big(\bigwedge_{\beta \in \mathcal{B}_\forall} \square\diamond\, moved_\beta \Big) \rightarrow \Big(\big(\bigwedge_{\beta \in \mathcal{B}_\exists} \square\diamond\, moved_\beta \big) \land \psi \Big),$$

which includes fresh Boolean variables $moved_\beta$ for every $\beta \in \mathcal{B}$. The intuition is that $moved_\beta$ holds whenever the stuttering β has performed a non-stuttering step in the last round. The LTL formula ψ_{mod} then requires that existentially quantified stutterings should progress infinitely often (expressed using the LTL operator combination $\square\diamond$), and ψ must hold, provided that all universally quantified stutterings (which will be in control of the refuter) are fair.

Let $\Theta \subseteq \mathcal{F}_{X_\mathcal{B}}$ be the *finite* set of first-order formulas used in ψ. For example, in φ_{OD}, $\Theta = \{l_{\beta_1} = l_{\beta_2}, o_{\beta_1} = o_{\beta_2}\}$. In the following, we assume that $\mathcal{A}_\psi = (Q_\psi, q_{0,\psi}, \delta_\psi, F_\psi)$ is a DBA over alphabet $2^{\Theta \cup \{moved_\beta \mid \beta \in \mathcal{B}\}}$ accepting exactly those infinite words that satisfy ψ_{mod}. That is, in each step, we need to tell $\mathcal{A}_\psi$'s transition function which of the formulas in Θ hold and which of the stutterings have been progressed (via the $moved_\beta$ variables), and $\mathcal{A}_\psi$ tracks if this behavior satisfies ψ_{mod}.

5.2 Game Vertices

In our asynchronous setting, different stutterings can point to (different positions on) the same trace. As outlined in Section 2, we accommodate this by maintaining a *window* of states, i.e., for each trace variable, we track a finite sequence of successive states in $\mathcal{T}$, akin to a sliding window.

$$\dfrac{\begin{array}{|l|}\hline \left\{s_\pi \in \kappa_\mathcal{T}\big(\Xi(\pi)(|\Xi(\pi)|-1)\big)\right\}_{\pi \in \mathcal{V}_\forall} \quad \textbf{(1.1)} \\ sched \subseteq \mathcal{B}_\forall \\ \hline \Xi' = \lambda \pi. \begin{cases} \Xi(\pi)\cdot s_\pi & \text{if } \pi \in \mathcal{V}_\forall \\ \Xi(\pi) & \text{otherwise} \end{cases} \quad \textbf{(1.2)} \\ \hline \end{array} \quad \begin{array}{|l|}\hline \mu' = \lambda \beta. \begin{cases} \mu(\beta)+1 & \text{if } \beta \in sched \\ \mu(\beta) & \text{otherwise} \end{cases} \quad \textbf{(1.3)} \\ \hline \flat' = \flat \cup sched \quad \textbf{(1.4)} \\ \hline \end{array}}{\left\langle \textcircled{\forall}, \Xi, \mu, \flat, q \right\rangle \rightsquigarrow \left\langle \textcircled{\exists}, \Xi', \mu', \flat', q \right\rangle}$$

Fig. 4. Transitions for vertices in the Ⓥ-stage.

Each stuttering variable β is then a (relative) pointer to the state window assigned to trace $base(\beta)$, i.e., the trace that β is a stuttering of. We parameterize our game with a bound $Z \in \mathbb{N}$, determining the maximal length of each state window. If we only deal with a single stuttering per trace (as is the case in most properties), a window of size $Z = 1$ generally suffices. If we consider properties with multiple stutterings on the same trace, the bound naturally defines a trade-off between the size of the game (a larger Z generates more vertices) and the flexibility of stutterings (a larger Z allows multiple stutterings of the same trace to diverge further). In many cases, we can statically infer a bound Z, while maintaining completeness (cf. Section 6).

We can now define our Büchi game $\mathcal{G}_{\mathcal{T},\varphi,Z}$ used to verify that $\mathcal{T} \models \varphi$. Each regular game vertex in $\mathcal{G}_{\mathcal{T},\varphi,Z}$ has the form $\langle stage, \Xi, \mu, \flat, q\rangle$, where **(1)** $stage \in \{\textcircled{\forall}, \textcircled{\exists}, \textcircled{\cup}\}$ tracks the current stage of the game; **(2)** $\Xi : \mathcal{V} \to (S_\mathcal{T})^{\leq Z+1}$ maps each trace variable to a window of states of length at most $Z + 1$; **(3)** $\mu : \mathcal{B} \to \{0, \dots, Z\}$ maps each stuttering variable β to a *relative* position within the window $\Xi(base(\beta))$; **(4)** $\flat \subseteq \mathcal{B}$ records which of the stutterings has made progress in the last step (used to evaluate $moved_\beta$); and **(5)** $q \in Q_\psi$ tracks the current state of $\mathcal{A}_\psi$. In addition to these regular game vertices, we add a dedicated error vertex v_{error}, which we will reach whenever the window bound Z is insufficient to accommodate all stutterings.

Example 5.1. Throughout this section, we will use our example from Figure 3 to illustrate our transition rules. In Figure 3, we represent a vertex $\langle stage, \Xi, \mu, \flat, q\rangle$ as follows: We depict $\Xi : \{\pi_1, \pi_2\} \to (S_\mathcal{T})^{\leq 2}$ as an array of nodes; depict the stuttering pointers $\mu : \{\beta_1, \beta_2\} \to \{0, 1\}$ as arrows; and color the stuttering pointer for β in thick iff $\beta \in \flat$. For example, the game vertex $\big\langle \textcircled{\cup}, \big[\pi_1 \mapsto [s_0, s_1], \pi_2 \mapsto [s_1, s_2]\big], [\beta_1 \mapsto 0, \beta_2 \mapsto 1], \{\beta_2\}, _\big\rangle$ is depicted as vertex v_{14}. △

We define our final Büchi game as $\mathcal{G}_{\mathcal{T},\varphi,Z} := (V_\mathfrak{V}, V_\mathfrak{R}, \alpha, F)$ where $V_\mathfrak{V}$ contains all vertices of the form $\langle \textcircled{\exists}, \Xi, \mu, \flat, q\rangle$, and $V_\mathfrak{R}$ contains all vertices of the form $\langle \textcircled{\forall}, \Xi, \mu, \flat, q\rangle$, $\langle \textcircled{\cup}, \Xi, \mu, \flat, q\rangle$, and the error vertex v_{error}. For the accepting states, we define $F := \{\langle stage, \Xi, \mu, \flat, q\rangle \mid q \in F_\psi\}$, i.e., the verifier wins a play by infinitely often visiting vertices in which the automaton state is accepting. An infinite play in $\mathcal{G}_{\mathcal{T},\varphi,Z}$ is thus won by the verifier iff the simulated run of $\mathcal{A}_\psi$ is accepting iff ψ_{mod} is satisfied on the infinite game-play. The transition function α is defined in Section 5.3.

5.3 Transition Rules

For the error vertex v_{error}, we define $\alpha(v_{error}) := \{v_{error}\}$. As v_{error} is a sink vertex that is not contained in F, a visit to v_{error} thus loses the game for the verifier. For each regular game vertex v, we define $\alpha(v) := \{v' \mid v \rightsquigarrow v'\}$, where $\rightsquigarrow$ is defined via the inference rules in Figures 4 to 6. Here, each inference rule adds a $\rightsquigarrow$-transition, provided the premise(s) (above the inference rule) are met.

Ⓥ-Stage. For each vertex $\langle \textcircled{\forall}, \Xi, \mu, \flat, q\rangle$ in the Ⓥ-stage, the refuter can progress all universally quantified traces and stutterings. The transitions from such vertices are defined in Figure 4. First,

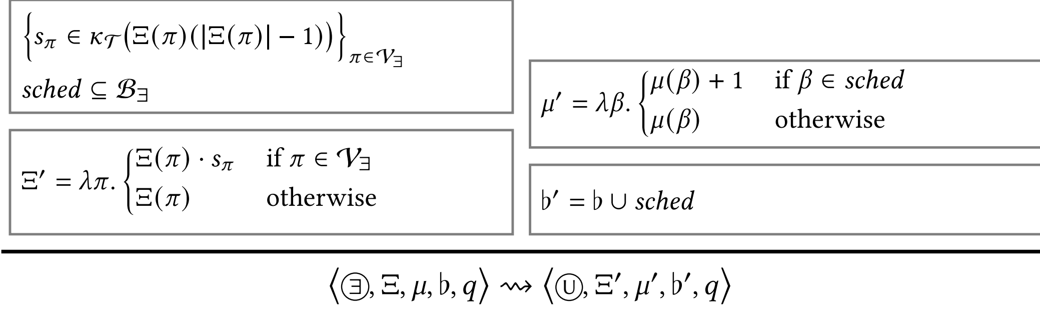

Fig. 5. Transitions for vertices in the ∃⃝-stage.

the refuter picks states $\{s_\pi \in \kappa_\mathcal{T}(\Xi(\pi)(|\Xi(\pi)|-1))\}_{\pi \in \mathcal{V}_\forall}$ for all universally quantified traces (premise **(1.1)**). Note how each s_π is a successor state in $\mathcal{T}$ of the *last* state in the state-window $\Xi(\pi)$ (i.e., $\Xi(\pi)(|\Xi(\pi)|-1)$), so each state window always models a continuous window along some path in $\mathcal{T}$. Additionally, the refuter can decide which of the universally quantified stutterings should be progressed by picking a subset $sched \subseteq \mathcal{B}_\forall$ (premise **(1.1)**). After $\{s_\pi\}_{\pi \in \mathcal{V}_\forall}$ and $sched$ are fixed, we can update the game vertex: We append s_π to the state window $\Xi(\pi)$ for each $\pi \in \mathcal{V}_\forall$ (premise **(1.2)**). At the same time, *sched* determines which (universally quantified) stuttering should be progressed (i.e., make a non-stuttering step by moving to the next position in the state window). For all $\beta \in sched$, we increment the $\mu(\beta)$ pointer (thus pointing to the next state in $base(\beta)$'s state window), and leave $\mu(\beta)$ unchanged for all other stutterings (premise **(1.3)**). We record which stutterings have progressed by setting $\flat' = \flat \cup sched$ (premise **(1.4)**). Note how each possible successor vertex of $\langle ⓥ, \Xi, \mu, \flat, q\rangle$ *precisely* corresponds to the intended actions that the refuter can take in each vertex (i.e., extend the trace windows of universally quantified traces and decide on universally quantified stutterings).

Example 5.2. In the example from Figure 3, the refuter only controls the trace variable π_1 (π_2, β_1, β_2 are existentially quantified). For example, vertex v_1 in Figure 3 represents game vertex $\langle ⓥ, [\pi_1 \mapsto [s_0], \pi_2 \mapsto [s_0]], [\beta_1 \mapsto 0, \beta_2 \mapsto 0], \emptyset, _\rangle$. According to the rules in Figure 4, this vertex has two successors. In premise **(1.1)**, we can either pick $s_{\pi_1} = s_0$ or $s_{\pi_1} = s_1$, the two possible successor states of s_0 (the last state in π_1's window), cf. the transition system $\mathcal{T}$ in Figure 2. This state is then appended to π_1'state window, leading to vertices v_2 and v_8, respectively. Note that the stuttering pointers are left unchanged as $\mathcal{B}_\forall = \emptyset$. △

∃⃝-Stage. Analogously, for vertices in the ∃⃝-stage, the verifier can progress all existentially quantified traces and stutterings, defined in Figure 5.

Example 5.3. Consider vertex v_9 in Figure 3 which represents game vertex $\langle ∃⃝, [\pi_1 \mapsto [s_0, s_1], \pi_2 \mapsto [s_1]], [\beta_1 \mapsto 0, \beta_2 \mapsto 0], \emptyset, _\rangle$. According to the rules in Figure 4, this vertex has four successors: The only choice for s_{π_2} (i.e., the state that is appended to π_2's state window) is $s_{\pi_2} = s_2$ (as this is the only successor of state s_1 in $\mathcal{T}$, cf. Figure 2). For the scheduling, the verifier can pick any $sched \subseteq \mathcal{B}_\exists = \{\beta_1, \beta_2\}$. Let us pick $sched = \{\beta_2\}$. According to the transition rules in Figure 5, we obtain vertex $\langle ⓤ, [\pi_1 \mapsto [s_0, s_1], \pi_2 \mapsto [s_1, s_2]], [\beta_1 \mapsto 0, \beta_2 \mapsto 1], \{\beta_2\}, _\rangle$, i.e., we append s_2 to π_2's state window, increment $\mu(\beta_2)$ by 1 (pointing to the next state in π_2 state window), leave $\mu(\beta_1)$ unchanged (so β_1 still points to the same state in π_1's state window as in the previous round, effectively stuttering β_1), and record that we have progressed β_2 by setting $\flat = \{\beta_2\}$. In Figure 3, this vertex is depicted as v_{14}. △

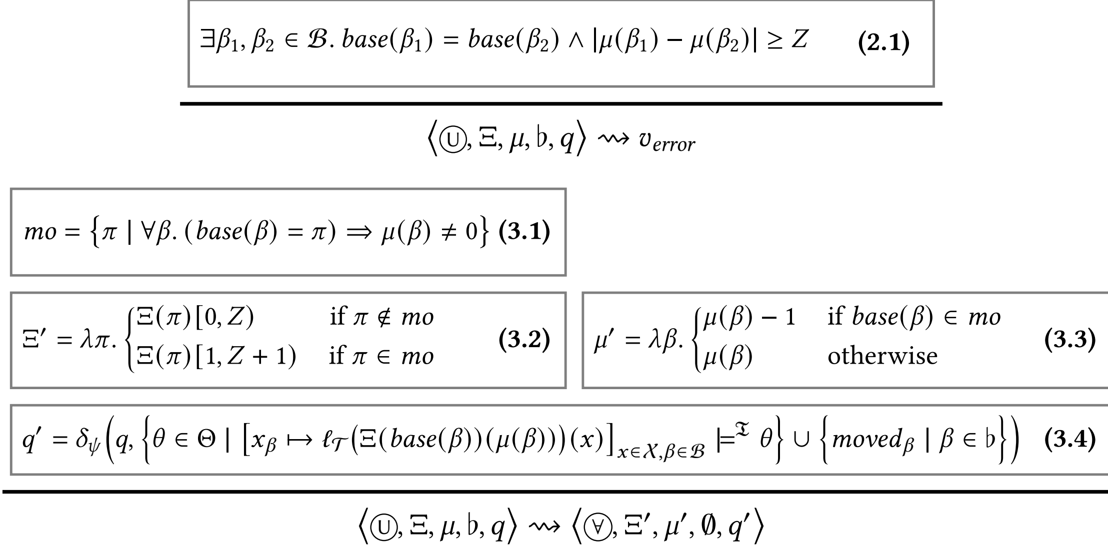

Fig. 6. Transitions for vertices in the Ⓤ-stage.

Ⓤ-Stage. The transitions of the Ⓤ-stage are defined in Figure 6. In **(2.1)**, we first check if the window size Z is insufficient to capture all stutterings for some trace variable, i.e., if there exist two stutterings β_1, β_2 that point to the same state window ($base(\beta_1) = base(\beta_2)$) and differ by at least Z steps. If this is the case, we move to v_{error} and thus let the verifier lose. Otherwise, we re-enter the Ⓐ-stage. Simultaneously, we update the automaton state of $\mathcal{A}_\psi$ (premise **(3.4)**): For this, we need to evaluate all first-order formulas $\theta \in \Theta$ (recall that $\Theta \subseteq \mathcal{F}_{X_\mathcal{B}}$ is the finite set of first-order formulas used as atoms in φ's LTL body ψ). Here, $\theta \in \Theta$ holds in the current game vertex iff $\big[x_\beta \mapsto \ell_\mathcal{T}\big(\Xi(base(\beta))(\mu(\beta))\big)(x)\big] \models^\mathfrak{T} \theta$. That is, for each indexed variable x_β, we take the current state for the β stuttering, which is the $\mu(\beta)$th state within $base(\beta)$'s state window, i.e., $\Xi(base(\pi))(\mu(\beta))$, and look up the value of x via $\mathcal{T}$'s labeling function $\ell_\mathcal{T}$. To evaluate the auxiliary propositions of the form $moved_\beta$ (as used by ψ_{mod} to ensure fair stutterings, cf. Section 5.1), we use $\flat$ and set $moved_\beta$ iff $\beta \in \flat$, i.e., iff β has been progressed in the previous Ⓐ-stage or Ⓔ-stage. Afterward, we reset $\flat$ to the empty set. In addition to updating the automaton state, we also trim the state windows in Ξ and update the relative pointers accordingly. For this, we first compute a set $mo \subseteq \mathcal{V}$, which contains all trace variables π where no stuttering indexes the first position, i.e., all stutterings β with $base(\beta) = \pi$ index a position greater than 0 (premise **(3.1)**). Premise **(3.2)** then shifts the windows of all traces in mo by removing their first position. By construction of mo, all stutterings have progressed past the first position, so the state at this position will never be used for the evaluation of ψ_{mod}. In addition to shifting the window, we also trim the end of the window to ensure that the window has length at most Z. For all stutterings $\beta \in mo$ where we shifted the state window (i.e., $base(\beta) \in mo$), we correct the relative positions in μ by decrementing the pointer (premise **(3.3)**). For every stuttering pointer β, the current state in $base(\beta)$'s state window thus does not change (we counteract all window shifts by decrementing the relative position).

Example 5.4. Consider vertex v_{14} in Figure 3, i.e., vertex $\big\langle Ⓤ, \big[\pi_1 \mapsto [s_0, s_1], \pi_2 \mapsto [s_1, s_2]\big], [\beta_1 \mapsto 0, \beta_2 \mapsto 1], \{\beta_2\}, _\big\rangle$. In premise **(3.1)**, we first compute $mo = \{\pi_2\}$. Note that π_1 is *not* in mo as β_1 points to π_1's first state, so we cannot drop the first position in π_1's window yet. We then, update the state windows in premise **(3.2)**: We remap π_1 to $[s_0, s_1][0, 1) = [s_0]$ (as $\pi_1 \notin mo$), and π_2 to $[s_1, s_2][1, 2) = [s_2]$ (recall that we set $Z = 1$). We correct for this, by decrementing $\mu(\beta_2)$ since $base(\beta_2) = \pi_2 \in mo$ (premise **(3.3)**). The stuttering β_2 thus still maps to the s_2 state in π_2's window

(which now is at position 0 in the updated state window). This update results in vertex $\langle ⓥ, [\pi_1 \mapsto [s_0], \pi_2 \mapsto [s_2]], [\beta_1 \mapsto 0, \beta_2 \mapsto 0], \emptyset, _\rangle$, depicted as vertex v_{15} in Figure 3. To progress the DBA (which we omitted in Figure 3), we need to evaluate the first-order formula in $\Theta = \{a_{\beta_1} \neq a_{\beta_2}\}$ (used as atomic formulas in φ_{fair}) and the propositions $moved_{\beta_1}$ and $moved_{\beta_2}$ (premise **(3.4)**). In vertex v_{14}, we obtain the current value of a_{β_1} (resp. a_{β_2}) by looking at the $\mu(\beta_1) = 0$th (resp. $\mu(\beta_2) = 1$th) position in state window $\Xi(base(\beta_1)) = \Xi(\pi_1) = [s_0, s_1]$ (resp. $\Xi(base(\beta_2)) = \Xi(\pi_2) = [s_1, s_2]$), which is state s_0 (resp. s_2), so $a_{\beta_1} = \ell_{\mathcal{T}}(s_0)(a) = 0$ (resp. $a_{\beta_2} = \ell_{\mathcal{T}}(s_2)(a) = 1$). Since $\flat = \{\beta_2\}$, we get $\{moved_\beta \mid \beta \in \flat\} = \{moved_{\beta_2}\}$, so we update the DBA state to $q' = \delta(q, \{a_{\beta_1} \neq a_{\beta_2}, moved_{\beta_2}\})$. Even though we omitted the DBA in Figure 3, it is easy to see that all paths in the strategy satisfy $\psi_{mod} = \bigcirc\square(a_{\beta_1} \neq a_{\beta_2}) \land \square\diamondsuit\, moved_{\beta_1} \land \square\diamondsuit\, moved_{\beta_2}$. △

5.4 Initial Vertices

Initially, we want to provide the verifier with as much information as possible, so we consider all possible state windows of length Z for universally quantified traces (cf. Section 6.2). Formally, we define V_{init} as all vertices $\langle ⓤ, \Xi, \mu, \emptyset, q_{0,\psi}\rangle$ where $\mu(\beta) = 0$ for all $\beta \in \mathcal{B}$, and the state windows in Ξ satisfy: $\Xi(\pi) = [s_{0,\mathcal{T}}]$ for all $\pi \in \mathcal{V}_\exists$ (i.e., all existentially quantified traces start in a state window of length 1), and for all $\pi \in \mathcal{V}_\forall$, $\Xi(\pi) = [s_0, s_1, \dots, s_{Z-1}]$ where $s_0 = s_{0,\mathcal{T}}$ and $s_{i+1} \in \kappa_{\mathcal{T}}(s_i)$ for $0 \leq i < Z - 1$ (i.e., all universally quantified traces start in a state window of length Z that contains consecutive states in $\mathcal{T}$ starting from $s_{0,\mathcal{T}}$). We are interested in checking if the verifier can win from all vertices in V_{init}, i.e., whether $wins_{\mathcal{G}_{\mathcal{T},\varphi,Z}}(\mathfrak{V}, V_{init})$.

5.5 Soundness

Theorem 5.5 (Soundness). *Consider any transition system $\mathcal{T}$, a $\forall^*\exists^*$ A-HLTL formula φ, and a bound $Z \in \mathbb{N}_{\geq 1}$. If $wins_{\mathcal{G}_{\mathcal{T},\varphi,Z}}(\mathfrak{V}, V_{init})$, then $\mathcal{T} \models \varphi$.*

Proof Sketch. We use a winning strategy for the verifier to construct witnesses for existentially quantified traces and stutterings. For this, we step-wise simulate the universal quantified traces and stutterings (taking the refuter's perspective) and use the strategy's response to construct witnesses. A full proof is given in the full version. □

Moreover, it is easy to see that our game-based approximation is monotone in the window size:

Lemma 5.6. *For any $\mathcal{T}, \varphi$, if $wins_{\mathcal{G}_{\mathcal{T},\varphi,Z}}(\mathfrak{V}, V_{init})$ and $Z' \geq Z$, then $wins_{\mathcal{G}_{\mathcal{T},\varphi,Z'}}(\mathfrak{V}, V_{init})$. For any $Z \in \mathbb{N}_{\geq 1}$, there exists $\mathcal{T}, \varphi$ such that $\neg wins_{\mathcal{G}_{\mathcal{T},\varphi,Z}}(\mathfrak{V}, V_{init})$ but $wins_{\mathcal{G}_{\mathcal{T},\varphi,Z+1}}(\mathfrak{V}, V_{init})$.*

In the example in Section 2, we dealt with a unique stuttering for each trace, so a window of size $Z = 1$ suffices to accommodate this single stuttering. If we consider *multiple* stutterings on the same trace, we sometimes need larger state windows and thus bounds $Z > 1$. We give a concrete example in the full version.

Remark 2. *In general, game $\mathcal{G}_{\mathcal{T},\varphi,Z}$ contains infinitely many states. If the system $\mathcal{T}$ is represented symbolically (e.g., using first-order logic), we can derive a symbolic description of $\mathcal{G}_{\mathcal{T},\varphi,Z}$ and either leverage existing approaches for solving infinite-state games [Beyene et al. 2014; de Alfaro et al. 2001; Farzan and Kincaid 2018; Heim and Dimitrova 2024; Laveaux et al. 2022; Samuel et al. 2021; Schmuck et al. 2024] or let the user solve the game interactively [Correnson and Finkbeiner 2025] (we discuss this further in Section 8). If $\mathcal{T}$ is a finite-state transition system (i.e., the set of states $S_{\mathcal{T}}$ is finite), $\mathcal{G}_{\mathcal{T},\varphi,Z}$ is a finite-state Büchi game, which we can compute directly. In this case, the number of vertices of $\mathcal{G}_{\mathcal{T},\varphi,Z}$ is exponential in the bound Z and the number of traces in φ (as is usual for self-compositions [Barthe et al. 2004]), but only* linear *in $|S_{\mathcal{T}}|$. As Büchi games can be solved in polynomial time [McNaughton 1993], checking if $wins_{\mathcal{G}_{\mathcal{T},\varphi,Z}}(\mathfrak{V}, V_{init})$ is decidable in polynomial time in the size of the system.*

6 Completeness

In general, our game-based verification approach is incomplete, i.e., the verifier might lose $\mathcal{G}_{\mathcal{T},\varphi,Z}$, even though $\mathcal{T} \models \varphi$ holds. In general, we thus cannot use the game to conclude the violation of a property; only prove satisfaction. This incompleteness is inevitable: For a finite-state system $\mathcal{T}$, checking if $wins_{\mathcal{G}_{\mathcal{T},\varphi,Z}}(\mathfrak{V}, V_{init})$ is decidable (cf. Remark 2), but $\mathcal{T} \models \varphi$ is, in general, undecidable [Baumeister et al. 2021]. However, our method is complete for many fragments and thus constitutes a (finite-state) decision procedure. In particular, the fragments for which our approach is complete subsume many previously known decidable classes [Baumeister et al. 2021; Hsu et al. 2023].

6.1 Alternation-Free Formulas

We first consider the case in which there is at most one stuttering per trace variable, and all traces and stutterings are quantified either existentially or universally. In this case, a window size of $Z = 1$ already ensures completeness:

Theorem 6.1. *Let $\mathcal{T}$ be any transition system, and let φ be an $\exists^*$ or $\forall^*$ A-HLTL formula with at most one stuttering per trace. Then $wins_{\mathcal{G}_{\mathcal{T},\varphi,1}}(\mathfrak{V}, V_{init})$ if and only if $\mathcal{T} \models \varphi$.*

6.2 Terminating Systems and Lookahead

In our game construction, we ensure that the state window of *universally* quantified traces is – already in the initial state – always of length Z. Recall that our game approximates the A-HLTL semantics: In the semantics of a $\forall^*\exists^*$ formula, all existentially quantified traces and stutterings are fixed knowing the *entire* universally quantified traces and stutterings. In our game, we let the verifier construct existentially quantified traces and stutterings step-wise by responding to the moves of the refuter. By always keeping the state window of universally quantifier traces of length Z (potentially including trailing states that are not pointed to by any stuttering), we thus offer limited clairvoyance to the verifier, i.e., the verifier can peek at the next Z states on universally quantified traces and make more informed decisions [Abadi and Lamport 1991]. This is particularly interesting in systems where only a fixed lookahead is needed to know the entire system execution. We say the transition system $\mathcal{T} = (S_\mathcal{T}, s_{0,\mathcal{T}}, \kappa_\mathcal{T}, \ell_\mathcal{T})$ is *terminating* (also called *tree-shaped* [Hsu et al. 2023]) if there exists a bound $D \in \mathbb{N}$ (called the *depth*), such that all paths of length at least D reach some sink state, i.e., a state $s \in S_\mathcal{T}$ with $\kappa_\mathcal{T}(s) = \{s\}$. Hsu et al. [2023] propose a QBF-based model-checking approach for A-HLTL in terminating systems. We can show that our approach is complete for terminating systems when choosing $Z = D$:

Theorem 6.2. *Let $\mathcal{T}$ be a terminating transition system with depth D and let φ be a $\forall^*\exists^*$ A-HLTL formula. Then $wins_{\mathcal{G}_{\mathcal{T},\varphi,D}}(\mathfrak{V}, V_{init})$ if and only if $\mathcal{T} \models \varphi$.*

6.3 Admissible Formulas

Next, we study the so-called *admissible* formulas proposed by Baumeister et al. [2021]. In the following, we assume that, in theory $\mathfrak{T}$, "=" is interpreted as equality on $\mathbb{V}$.

Definition 6.3. An admissible formula has the form $\forall \pi_1 \ldots \forall \pi_n. \exists \beta_1 \blacktriangleright \pi_1 \ldots \exists \beta_n \blacktriangleright \pi_n. \psi$, where ψ is a boolean combination of: **(1)** Any number of *state formulas*, i.e., formulas that use no temporal operators; and **(2)** A *single* (positively occurring) *phase formula* of the form $\bigwedge_{i<j} \Box \big(\bigwedge_{a \in P_{i,j}} (a_{\beta_i} = a_{\beta_j}) \big)$, where $P_{i,j} \subseteq \mathcal{X}$. △

That is, we existentially quantify over a *unique* stuttering for each trace and limit the relational formulas allowed in ψ. We can think of the set $P_{i,j}$ as denoting a set of *colors*, defined by the evaluation of variables in $P_{i,j}$ (so there are $|\mathbb{V}|^{|P_{i,j}|}$ many colors). A phase formula then asserts

$\square(\bigwedge_{a\in P_{i,j}} a_{\beta_i} = a_{\beta_j})$ for each pair $i < j$ of traces, and thus requires that π_i and π_j traverse the same sequence of $P_{i,j}$-colors, albeit at different speed (via stutterings β_i and β_j, respectively). For example, φ_{OD} is admissible (see [Baumeister et al. 2021] for further examples). We can show that our approach is complete for admissible formulas (already for $Z = 1$):

Theorem 6.4. *Let $\mathcal{T}$ be any transition system, and let φ be an admissible A-HLTL formula. Then $wins_{\mathcal{G}_{\mathcal{T},\varphi,1}}(\mathfrak{V}, V_{init})$ if and only if $\mathcal{T} \models \varphi$.*

In the following, we provide a proof outline of Theorem 6.4. The first direction follows from Theorem 5.5. For the other, assume φ is admissible, and $\mathcal{T} \models \varphi$. Let $\psi_{phase} = \bigwedge_{i<j} \square\big(\bigwedge_{a\in P_{i,j}}(a_{\beta_i} = a_{\beta_j})\big)$ be the unique phase formula in ψ. The objective of the verifier is thus to find stutterings that satisfy ψ_{phase} (whenever such a stuttering exists).

Safe Progress Set. Each vertex v in $\mathcal{G}_{\mathcal{T},\varphi,1}$ controlled by the verifier $\mathfrak{V}$ has the form

$$v = \big\langle ⓔ, \big[\pi_i \mapsto [s_i, s_i']\big]_{i=1}^{n}, \big[\beta_i \mapsto 0\big]_{i=1}^{n}, \emptyset, q\big\rangle,$$

i.e., the traces $\pi_1, \dots, \pi_n$ are mapped to length-2 state windows $[s_1, s_1'], \dots, [s_n, s_n']$, respectively, and all stutterings $\beta_1, \dots, \beta_n$ point to the 0th position in the windows. We assume that v locally satisfies all coloring constraints in ψ_{phase}, i.e., for all $i < j$ and $a \in P_{i,j}$, $\ell_{\mathcal{T}}(s_i)(a) = \ell_{\mathcal{T}}(s_j)(a)$; if this is not the case, ψ_{phase} is already violated. Call this assumption **(A)**. In vertex v, the verifier can decide on which of the stutterings $\beta_1, \dots, \beta_n$ should be progressed. We can thus identify each possible successor vertex of v by a so-called *progress set* $M \subseteq \{1, \dots, n\}$. Intuitively, M contains exactly those indices i on which β_i performs a non-stuttering step by progressing from state s_i to s_i'. For $M \subseteq \{1, \dots, n\}$, we define states $next_v^M(1), \dots, next_v^M(n) \in S_{\mathcal{T}}$ by defining $next_v^M(i) := s_i'$ if $i \in M$ and otherwise as $next_v^M(i) := s_i$. That is, every stuttering β_i progressed in M is mapped to the second position in the state window in v (i.e., s_i'), and all non-progressed stutterings remain in the first state (i.e., s_i). We say progress set M is *safe* if for every $i < j$, and every $a \in P_{i,j}$, $\ell_{\mathcal{T}}(next_v^M(i))(a) = \ell_{\mathcal{T}}(next_v^M(j))(a)$, i.e., M ensures that all coloring constraints are satisfied *locally* in the next step.

Maximal Safe Progress Set. Crucially, the coloring constraints in ψ_{phase} are equalities and thus symmetric: For example, assume that **(1)** progress set $\{i\}$ is safe, and **(2)** progress set $\{j\}$ is also safe. Then, by **(1)**, s_i' has the same $P_{i,j}$-color as s_j, by **(2)**, s_i has the same $P_{i,j}$-color as s_j', and, by **(A)**, s_i has the $P_{i,j}$-same color as s_j. This already implies that s_i' has the same $P_{i,j}$-color as s_j'; progress set $\{i, j\}$ is also safe. More generally, the set of safe progress sets forms a join-complete semilattice:

Lemma 6.5. *If M_1 and M_2 are safe progress sets, then $M_1 \cup M_2$ is a safe progress set.*

We can thus define M_v^{max} as the unique maximal safe progress set in vertex v (defined as the union of all safe progress sets). We can now, informally, define a winning strategy σ^{max} for $\mathfrak{V}$: In each vertex v that satisfies **(A)** and still needs to satisfy ψ_{phase}, we define $\sigma^{max}(v) := v'$, where

$$v' := \big\langle ⓤ, \big[\pi_i \mapsto [s_i, s_i']\big]_{i=1}^{n}, \big[\beta_i \mapsto ite(i \in M_v^{max}, 1, 0)\big]_{i=1}^{n}, \{\beta_i \mid i \in M_v^{max}\}, q\big\rangle.$$

Here, we write $ite(b, x, y)$ to be x if b holds and otherwise y (short for if-then-else). In v', the verifier advances the β_i-stuttering on π_i (by incrementing the μ-pointer from position 0 to 1) iff $i \in M^{max}$. If v does not need to satisfy ψ_{phase} (e.g., because the state formulas already suffice to satisfy ψ), the verifier progresses all stutterings $\beta_1, \dots, \beta_n$.

Intuitively, σ^{max} thus always (locally) progresses as many stutterings as possible while ensuring that the coloring constraints in ψ_{phase} hold in the next step. For any combination of traces $\pi_1, \dots, \pi_n$ that satisfies $\exists \beta_1 \blacktriangleright \pi_1 \dots \exists \beta_n \blacktriangleright \pi_n.\, \psi_{phase}$, strategy σ^{max} automatically finds a stuttering such that ψ_{phase} holds. It is thus easy to see that the verifier wins $\mathcal{G}_{\mathcal{T},\varphi,1}$ by following strategy σ^{max}, proving Theorem 6.4.

6.4 A New Decidable Fragment

Using a similar idea as in Section 6.3, we can show that our game-based method is complete for an even broader class of properties. The main idea is that we want to preserve the ability to find a maximal safe progress set, i.e., whenever we can safely progress some stuttering β_i and can safely progress some stuttering β_j, we can also safely progress both β_i and β_j.

Given an assignment $A : \mathcal{X} \to \mathbb{V}$, and a stuttering variable $\beta \in \mathcal{B}$, we write $A^\beta : \mathcal{X}_{\{\beta\}} \to \mathbb{V}$ for the indexed assignment defined as $A^\beta(x_\beta) := A(x)$. For two stuttering variables $\beta_i, \beta_j \in \mathcal{B}$, we say a formula $\theta \in \mathcal{F}_{\mathcal{X}_{\{\beta_i,\beta_j\}}}$ is *rectangle closed* if the following holds: for any assignments $A, B, C, D : \mathcal{X} \to \mathbb{V}$ we have:

$$\left(A^{\beta_i} \cup B^{\beta_j} \models^{\mathfrak{I}} \theta\right) \land \left(C^{\beta_i} \cup B^{\beta_j} \models^{\mathfrak{I}} \theta\right) \land \left(A^{\beta_i} \cup D^{\beta_j} \models^{\mathfrak{I}} \theta\right) \Rightarrow \left(C^{\beta_i} \cup D^{\beta_j} \models^{\mathfrak{I}} \theta\right),$$

i.e., if (A, B), (C, B), and (A, D) all satisfy θ, then so does (C, D).

In admissible formulas, we always deal with equalities of the form $a_{\beta_i} = a_{\beta_j}$ (for $a \in P_{i,j}$), which are clearly rectangle closed. More generally, all equalities of the form $\theta = (e_1 = e_2)$ – where e_1 and e_2 are first-order terms (cf. [Barwise 1977]) over a *unique* (not necessarily the same) stuttering variable – are rectangle closed. For example, $a_{\beta_1} = (b_{\beta_2} + c_{\beta_2})$ is rectangle closed, but $a_{\beta_1} = (b_{\beta_2} + c_{\beta_3})$ is not.

Definition 6.6. A *rectangle closed invariant* is a formula $\varphi = \forall \pi_1 \dots \pi_n. \exists \beta_1 \blacktriangleright \pi_1 \dots \exists \beta_n \blacktriangleright \pi_n. \psi$, where ψ is a boolean combination of: **(1)** Any number of state formulas; and **(2)** A *single* (positively occurring) formula $\square \bigwedge_{\theta \in \Theta} \theta$ where every $\theta \in \Theta$ is rectangle closed. △

It is easy to see that any admissible formula is a rectangle closed invariant (any equality $a_{\beta_i} = a_{\beta_j}$ is rectangle closed). Note that an admissible formula only allows constraints of the form $a_{\beta_i} = a_{\beta_j}$, i.e., assert equalities between the *same* variable on both traces. Rectangle closed invariants allow arbitrary expressions and can thus relate *arbitrary* variables between two traces. Using the same proof idea as for Theorem 6.4, we can show the more general result:

THEOREM 6.7. *Let $\mathcal{T}$ be any transition system, and let φ be a rectangle closed invariant. Then $wins_{\mathcal{G}_{\mathcal{T},\varphi,1}}(\mathfrak{V}, V_{init})$ if and only if $\mathcal{T} \models \varphi$.*

Example 6.8. For example, assume that a system consists of bits (Boolean variables) $w1, \dots, wn$, but an attacker cannot observe the bits individually. Instead, using a probing attack [Wang et al. 2017], it can detect if *some* bit is currently enabled. We can generalize Zdancewic and Myers [2003]'s observational determinism to this more restrictive attacker setting by defining

$$\forall \pi_1.\forall \pi_2.\exists \beta_1 \blacktriangleright \pi_1.\exists \beta_2 \blacktriangleright \pi_2.\, (l_{\beta_1} = l_{\beta_2}) \to \square\left((w1_{\beta_1} \lor \dots \lor wn_{\beta_1}) = (w1_{\beta_2} \lor \dots \lor wn_{\beta_2})\right),$$

requiring that only the value of $w1 \lor \dots \lor wn$ agrees on both traces. △

7 Implementation and Evaluation

Beyond our theoretical contributions, our game can be used directly for the (semi-) automated verification of A-HLTL. Notably, our entire framework applies to both finite and infinite-state systems. For infinite-state systems, we envision our approach to be used for interactive verification (similar to successful approaches for HyperLTL [Correnson and Finkbeiner 2025]); we discuss this further in Section 8. For finite-state systems, we can directly construct and solve the game, resulting in a fully automated verification method. Note that such finite-state methods are still interesting in an infinite-state setting. Abstractions (e.g., generated by a set of predicates [Graf and Saïdi 1997] or an abstract domain [Cousot and Cousot 1977]) typically abstract infinite variable domains, while maintaining the temporal behavior of the system, i.e., the abstraction of an infinite-state system results in a finite-state system where each step of the infinite-state system corresponds to one

Table 1. We compare HyMCA with HyperQB [Hsu et al. 2023] on terminating systems. We depict the size of the system(s) ($|S|$), the verification result, and the total time taken by each tool (t). For HyMCA, we additionally give the time needed to construct (t_{const}) and solve (t_{solve}) $\mathcal{G}_{\mathcal{T},\varphi,1}$, as well as the number of vertices ($|\mathcal{G}|$). Execution time is measured in seconds, and the timeout (denoted "-") is set to 5 minutes. Note that for HyMCA, the total time t can be slightly larger than the sum of t_{const} and t_{solve}, as it, e.g., includes parsing and preprocessing.

				[Hsu et al. 2023]	HyMCA			
Instances	**Property**	$\|S\|$	**Result**	t	t_{const}	$\|\mathcal{G}\|$	t_{solve}	t
ACDB	φ_{OD}	110	✗	2.96	1.00	17,119	0.54	**1.86**
ACDB$_{ndet}$	φ_{OD}	679	✗	12.06	3.58	71,269	2.29	**7.18**
ConcLeak	φ_{OD}	577	✗	27.16	3.44	59,495	2.29	**6.51**
ConcLeak$_{ndet}$	φ_{OD}	2821	✗	-	62.1	899,014	45.9	**114.4**
SpecExec$_{V1}$	φ_{SNI}	29/57	✗	3.71	0.199	1,219	0.09	**0.69**
SpecExec$_{V2}$	φ_{SNI}	109/169	✓	8.15	0.15	355	0.05	**0.82**
SpecExec$_{V3}$	φ_{SNI}	88/169	✗	11.65	0.38	4,041	0.19	**1.69**
SpecExec$_{V4}$	φ_{SNI}	94/169	✗	10.45	0.71	11,797	0.34	**2.01**
SpecExec$_{V5}$	φ_{SNI}	94/169	✗	7.91	0.15	925	0.05	**0.96**
SpecExec$_{V6}$	φ_{SNI}	85/169	✗	10.80	0.62	9,573	0.29	**1.93**
SpecExec$_{V7}$	φ_{SNI}	109/169	✓	7.65	0.13	355	0.04	**0.82**
DBE	φ_{SC}	9/7	✓	1.22	0.13	469	0.06	**0.43**
LP	φ_{SC}	81/77	✓	3.22	0.78	18,705	0.63	**1.63**
EFLP	φ_{SC}	81/249	✓	15.34	2.6	68,015	2.4	**5.82**
CacheTA	φ_{OD}	49	✗	2.51	0.44	7,683	0.34	**1.03**
CacheTA$_{ndet+loops}$	φ_{OD}	59	✗	**3.79**	2.26	72,812	2.54	5.07
CacheTA$_{ndet+loops}$	φ_{OD}	87	✗	23.31	6.26	170,509	6.9	**13.57**

computation step of the finite-state abstraction. Verifying properties like Zdancewic and Myers [2003]'s OD on the abstraction, therefore, still requires finding an appropriate stuttering. The core computational challenge of A-HLTL verification is still present in the finite-state abstraction, making finite-state verification results and tools relevant.

As a proof-of-concept, we have implemented our game-based method for finite-state systems in a tool we call HyMCA. HyMCA reads a system $\mathcal{T}$ (in the form of a symbolic NuSMV system [Cimatti et al. 2002]), an A-HLTL formula φ, and a bound $Z \in \mathbb{N}$, and automatically constructs and solves $\mathcal{G}_{\mathcal{T},\varphi,Z}$ (encoded as an explicit-state game). If desired, HyMCA can also resolve traces on *different* systems [Goudsmid et al. 2021]. Internally, we use spot [Duret-Lutz et al. 2022] to convert LTL formulas to deterministic automata and use oink [van Dijk 2018] to solve $\mathcal{G}_{\mathcal{T},\varphi,Z}$.

Terminating Systems. First, we evaluate HyMCA on terminating systems using the benchmarks from Hsu et al. [2023]. While the QBF-based bounded model-checking approach of Hsu et al. [2023] is, in theory, applicable to arbitrary formulas, their implementation (HyperQB) only implements a few fixed formula templates. We use the following subset of admissible templates: observational determinism (φ_{OD}), speculative non-interference [Guarnieri et al. 2020] (φ_{SNI}), and correct compilation (φ_{SC}), and check them on various NuSMV models. See [Hsu et al. 2023] for details. All three properties are admissible and thus fall in the fragment for which HyMCA is complete (cf. Theorem 6.4).

Table 2. We verify A-HLTL properties on non-terminating (reactive) systems. We give the verification result and execution time of HyMCA.

System	Property	Result	t_{HyMCA}
Figure 1a$_{syn}$ (2 bit)	φ_{OD}	✓	0.31
Figure 1a$_{syn}$ (4 bit)	φ_{OD}	✓	0.45
Figure 1a$_{syn}$ (8 bit)	φ_{OD}	✓	1.21
Figure 1a (2 bit)	φ_{OD}	✓	0.33
Figure 1a (4 bit)	φ_{OD}	✓	1.15
Figure 1a (8 bit)	φ_{OD}	✓	109.43

System	Property	Result	t_{HyMCA}
Figure 2	φ_{OD}	✓	0.25
Buffer	φ_{OD}	✗	0.46
Buffer	φ_{NI}	✓	0.54
Buffer$_{delay,Z=3}$	φ_{NI}	✓	31.63
Buffer$_{flipped}$	φ_{OD}	✗	0.76
Buffer	φ_{NI}	✓	1.32

We depict the verification results and times in Table 1. We used the fixed window size of $Z = 1$, which suffices for completeness. We write ✓(resp. ✗) if the property is satisfied (resp. violated), which – by completeness – we can directly infer by solving the game. Generally, we observe that the HyperQB performs well if the system contains many states but is very shallow, leading to a small QBF encoding. Still, HyMCA performs faster than HyperQB in a majority of the instances.

Reactive Systems. A particular strength of our approach is the ability to, for the first time, automatically verify A-HLTL on reactive, i.e., non-terminating, systems. We depict a few test cases in Table 2. First, we check φ_{OD} on the programs from Figure 1a (and a synchronous version thereof). To increase the size of the system, we scale the number of bits stored in each variable. We also check more complex (non-admissible) A-HLTL properties like φ_{NI} (cf. Example 4.1) on non-terminating systems. For example, the Buffer instance models a system that propagates the low-security input (which can change in each step) to the output (potentially with a delay). The system, therefore, violates φ_{OD} but satisfies φ_{NI}. We stress that these examples are *not* meant as real-world examples but rather serve as simple abstractions of real-world components. The A-HLTL properties reason about the infinite executions of these programs, making verification very challenging (none of the previous methods could automatically verify these examples).

Beyond Finite-State Systems. We emphasize that HyMCA implements our novel game-based verification approach in its simplest possible form by computing an *explicit-state* parity game. Our preliminary experiments show that even such a direct implementation performs well compared to previous QBF-based methods. We stress that the main complexity in A-HLTL verification stems from the expressiveness of the logic itself; the complexity (size) of the system plays a secondary role. All properties checked in Tables 1 and 2 use the high-level asynchronous reasoning of A-HLTL, and delegate the search for an appropriate asynchronous stuttering to the verification tool (cf. Section 8). Our prototype is not meant as a full-fledged verification tool. Instead, it demonstrates that the game-based approach works well in a *finite-state setting*. A key advantage of our game lies in its applicability to infinite-state systems. For example, in the domain of *synchronous* HyperLTL, Coenen et al. [2019b] showed that the synchronous game-based approach works well in a finite-state setting, leading to successful adoption to infinite-state systems; either by solving infinite-state games automatically [Beutner and Finkbeiner 2022b; Itzhaky et al. 2024] or interactively [Correnson and Finkbeiner 2025]. Our approach handles more general asynchronous properties (including well-known examples like OD), and provides a valuable abstraction: instead of reasoning over infinite traces and stutterings, we only need to reason *locally* over a finite window of states and pointers within that window. As we argue in Section 8, our game thus forms the foundation to extend verification to infinite-state systems.

8 Related Work

Relational Program Logics. *Relational Hoare Logic* [Benton 2004] – and related Hoare-style hyperproperty specifications [Assaf et al. 2017; Beutner 2024; Dardinier and Müller 2024; Dickerson et al. 2022; D'Osualdo et al. 2022; Gladshtein et al. 2024; Maillard et al. 2020; Sousa and Dillig 2016] – relate the initial and final states of *multiple* program runs and are thus inherently "*asynchronous*" (program executions can take a different number of steps to termination). However, RHLs struggle to express *temporal* hyperproperties, i.e., properties that reason about the *temporal behavior* along possibly infinite executions (as, e.g., found in infinite protocol interactions or reactive systems), which are needed for properties like Zdancewic and Myers [2003]'s observational determinism.

Hypersafety and Commutativity. Various works have studied the verification of hypersafety properties by exploiting commutativity [Antonopoulos et al. 2023; Eilers et al. 2023, 2020; Farzan et al. 2022; Farzan and Vandikas 2020; Shemer et al. 2019]. These approaches attempt to find an alignment of different executions that aids verification by exploiting the fact that we can interleave different program executions. Our work differs from these approaches in that *quantification over alignments (aka stutterings) is part of the specification* and not just a technique that helps during verification. Let us take Zdancewic and Myers [2003]'s observational determinism (OD) as an example, which states that all pairs of traces with (initially) identical low-security input are stutter-equivalent w.r.t. the output. To verify this property, intermediate program steps, i.e., steps where the output does not change, can be interleaved arbitrarily. The problem is that one of the verification challenges lies in the *identification* of points where the output changes.

```
1 o = 0
2 c = 0
3 repeat
4   o = ⌊ c / h ⌋
5   c = c + 1
```

Fig. 7. Example program

Example 8.1. Consider the simple integer-based program in Figure 7. Here, $\lfloor \cdot \rfloor$ rounds down to the nearest integer. It is easy to see that the output sequence of this program (per loop iteration) is $0^h 1^h 2^h \cdots$, i.e., for the first h loop iterations, the value of o is 0, and so forth. The output of the program is stutter-equivalent independent of h; the program satisfies OD. Yet, verifying this using commutativity-based frameworks is difficult. Most existing frameworks [Beutner and Finkbeiner 2022b; Eilers et al. 2020; Farzan and Vandikas 2020; Itzhaky et al. 2024; Shemer et al. 2019; Unno et al. 2021] search for *some* alignment that facilitates an easy proof of the property. To express OD, we do, however, want to compare the output of two executions *whenever the output changes*, i.e., identifying points at which the executions should synchronize is (implicitly) part of the OD specification. In the above frameworks, we would need to enforce the alignment manually (by, e.g., defining explicit observation points or adding synchronization assertions). Only *afterward* can the k-safety verifier find some alignment (between two points where the output changes) that aids verification. In our example, we cannot infer the points where the output changes syntactically. In contrast, A-HLTL offers a richer, higher-level specification language, allowing us to easily express Zdancewic and Myers [2003]'s OD by using first-class *quantification* over stutterings (see φ_{OD}), i.e., the identification of points that need to be aligned is *part of the specification itself.* △

Our approach thus differs from existing verification approaches in the expressiveness of the specification language: Most hypersafety frameworks exclusively focus on the verification challenges stemming from infinite state space, whereas our work focuses on the complexity stemming from the specification logic itself.

Temporal Logics for Asynchronous Hyperproperties. In recent years, many logics have been proposed for the specification of asynchronous hyperproperties Examples include A-HLTL (the main object of study in this paper) [Baumeister et al. 2021], H_μ and mumbling H_μ (extensions of the

polyadic μ-calculus), [Gutsfeld et al. 2021, 2024], OHyperLTL [Beutner and Finkbeiner 2022b] (an extension of HyperLTL with explicit observation points), HyperLTL$_S$ [Bozzelli et al. 2021] (an extension of HyperLTL where operators can skip evaluation steps), HyperLTL$_C$ [Bozzelli et al. 2021] (an extension of HyperLTL where only some traces progress), variants of team semantics [Gutsfeld et al. 2022; Krebs et al. 2018], and extensions of first-order logic with explicit quantification over time [Bartocci et al. 2022]. Bozzelli et al. [2022] studied the expressiveness of many of these logics and found that most of them are *incomparable*.

Verification of A-HLTL. Baumeister et al. [2021] study a restricted class of A-HLTL properties called *admissible* (see Section 6.3). They show that finite-state model-checking of admissible formulas is decidable, and provide a manual reduction to (synchronous) HyperLTL. Hsu et al. [2023] study *terminating* systems (see Section 6.2) and present a QBF-based bounded model-checking approach. Unlike previous techniques for A-HLTL, our method is soundly applicable to arbitrary $\forall^*\exists^*$ A-HLTL formulas, i.e., it might succeed in verifying a property that does not fall into any known decidable fragment.

$\forall^*\exists^*$ *and Game-Based Verification.* In recent years, many techniques for verifying hyperproperties *beyond* k-safety have been proposed [Antonopoulos et al. 2023; Beutner and Finkbeiner 2022b; Itzhaky et al. 2024; Unno et al. 2021]. These techniques combine the search for a suitable alignment with the search for witness traces for existential quantifiers. For example, Unno et al. [2021] encode verification of $\forall^*\exists^*$ properties as a specialized form of Horn clauses and let the solver figure out an appropriate alignment. Currently, their encoding only searches for *some* alignment that aids verification. It is an interesting future work to study if the encoding can be extended with *explicit quantification* over alignments (as in A-HLTL). Beutner et al. [2024]; Correnson et al. [2024]; Hsu et al. [2023] employ bounded unrolling to verify or refute $\forall\exists$ properties. Our verification approach is rooted in a game-based interpretation of traces and stutterings. For *synchronous* HyperLTL properties, such game-based interpretations have been employed successfully to approximate expensive quantifier alternations [Beutner and Finkbeiner 2022a; Coenen et al. 2019b]. The dynamics of our asynchronous game differ substantially from synchronous games. Our key idea of using *windows* and *relative pointers* allows the players to have direct control over the stuttering of traces, which is key to approximating A-HLTL's semantics. In particular, our asynchronous game allows the players to control the speed (i.e., stuttering) of the traces *individually*. Beutner and Finkbeiner [2022b]; Itzhaky et al. [2024] employ a game-based semantics for asynchronous $\forall^*\exists^*$ properties expressed in OHyperLTL. In this logic, the *user* indicates at which points the executions should synchronize, and the verifier attempts to find an alignment between these so-called *observation points*. As argued before, A-HLTL allows explicit *quantification* over stutterings, which is necessary to succinctly express properties like OD; to express OD in OHyperLTL, we would need to manually place observation points whenever the output changes, which is often not possible syntactically (cf. Example 8.1). It is interesting to see if the CHC encoding by Itzhaky et al. [2024] can be extended to allow explicit reasoning over stutterings, perhaps using our novel idea of using state windows and relative pointers.

Refinement, Stuttering, and Simulation. A particularly interesting class of $\forall\exists$ hyperproperties are *refinement* properties. Given two systems $\mathcal{T}_1$ and $\mathcal{T}_2$, and a set of observable variables *Obs*, we can easily express a lock-step refinement property using (synchronous) HyperLTL as $\varphi^{syn}_{refine} := \forall\pi_1.\exists\pi_2.\square\big(\bigwedge_{a\in Obs} a_{\pi_1} = a_{\pi_2}\big)$. A popular method for *proving* a refinement property (such as φ^{syn}_{refine}) is to establish a *simulation relation* between $\mathcal{T}_1$ and $\mathcal{T}_2$ [Milner 1971]. A popular technique for automatically *checking* the existence of a (synchronous) simulation relation is to exploit the connection between simulation relations and winning strategies in *simulation games* [Stirling 1995].

While φ^{syn}_{refine} requires a lock-step refinement, it often suffices to establish a *stuttering* refinement property (see, e.g., [Leroy 2006]), which we can easily express in A-HLTL as

$$\forall \pi_1. \exists \pi_2. \exists \beta_1 \blacktriangleright \pi_1. \exists \beta_2 \blacktriangleright \pi_2. \square \Big(\bigwedge_{a \in Obs} a_{\beta_1} = a_{\beta_2} \Big). \qquad (\varphi_{refine})$$

That is, for any execution π_1 in $\mathcal{T}_1$, there exists some execution π_2 in $\mathcal{T}_2$ that, *up to stuttering*, agrees on the values of all observable variables. We can prove φ_{refine} by finding a *stutter simulation* [Namjoshi 1997; Namjoshi and Tabajara 2020]: As long as either of the systems makes a "tau-step", i.e., a step that does not change any variable in *Obs* and thus does not emit a "visible event", the other system is allowed to stutter. Our game-based technique can thus be used as a principled method for proving stuttering refinement. In this light, we can see our game-based verification method as an asynchronous extension of synchronous simulation games; at least in the very special case of φ_{refine}. Notably, our game-based method – when applied to φ_{refine} – naturally allows both copies to be stuttered independently (as long as the stuttering is progressed *infinitely often* on both traces), i.e., our technique does not require *synchronous progress* where both systems eventually progress *together* (cf. [Cho et al. 2023]).

Interactive Verification of Infinite-State Systems. In our implementation, we focused on finite-state systems in order to evaluate our method against existing approaches. All our results (including soundness and completeness) also apply to infinite-state systems. While we could attempt to construct and solve the resulting infinite-state game automatically (see, e.g., [Beyene et al. 2014; de Alfaro et al. 2001; Farzan and Kincaid 2018; Heim and Dimitrova 2024; Laveaux et al. 2022; Samuel et al. 2021; Schmuck et al. 2024]), the more promising direction is to aim for an interactive proof method. Correnson and Finkbeiner [2025] recently used the game-based approach for *synchronous* HyperLTL [Beutner and Finkbeiner 2022a; Coenen et al. 2019b] to develop an *interactive* proof system to verify ∀∃ properties in infinite-state software. Intuitively, their approach uses co-induction to incrementally unfold the underlying game and let the user implicitly define a strategy. Our game-based approach can handle asynchronous hyperproperties that quantify over stutterings, but still yields a plain two-player game. This game enables a clear verification objective for the user and provides the much-needed abstraction from infinite traces and stutterings; the user only needs to reason about stuttering in the current state. Our game then allows us to employ frameworks similar to those used by Correnson and Finkbeiner [2025] to let the user interactively prove A-HLTL properties by incrementally unfolding the game arena. Our work thus lays the foundation for developing a unified proof system that can handle many prominent information-flow policies in reactive systems.

9 Conclusion and Future Work

In this paper, we have presented a principled approach for the (automated or interactive) verification of asynchronous hyperproperties expressed in A-HLTL. Our method approximates quantifier alternation as a game and seamlessly supports quantification over stutterings. This allows for sound verification of arbitrary $\forall^*\exists^*$ A-HLTL properties, well beyond the fragments supported by previous methods. Moreover, we have shown that our method is complete for many fragments and thus constitutes a finite-state decision procedure.

Our work leaves numerous exciting directions for future extension. On the theoretical side, one can, e.g., extend our game-based view to support A-HLTL formulas beyond the $\forall^*\exists^*$ fragment by incorporating incomplete information [Reif 1984]; intuitively, the verifier should not base its decision on universal traces and stutterings quantified *after* the (existentially quantified) objects controlled by the verifier [Beutner and Finkbeiner 2025b,c]. On the practical side, we consider the

verification of infinite-state (software) systems to be of particular interest. As we argued above, our game-based methods (and proofs) also apply to (symbolically represented) *infinite-state* systems, yielding infinite-state games. Exploring effective techniques to solve the resulting (now infinite-state) game and developing abstraction techniques for a (possibly unbounded) window of states are challenging open problems. More concretely, it is interesting to apply the interactive proof system of Correnson and Finkbeiner [2025] to our game, allowing the user to verify asynchronous hyperproperties in infinite-state reactive systems interactively.

Acknowledgments

This work was supported by the European Research Council (ERC) Grant HYPER (101055412), and by the German Research Foundation (DFG) as part of TRR 248 (389792660).

References

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* (1991). doi:10.1016/0304-3975(91)90224-P

Erika Ábrahám, Ezio Bartocci, Borzoo Bonakdarpour, and Oyendrila Dobe. 2020. Probabilistic Hyperproperties with Nondeterminism. In *International Symposium on Automated Technology for Verification and Analysis, ATVA 2020s.* doi:10.1007/978-3-030-59152-6_29

Erika Ábrahám and Borzoo Bonakdarpour. 2018. HyperPCTL: A Temporal Logic for Probabilistic Hyperproperties. In *International Conference on Quantitative Evaluation of Systems, QEST 2018.* doi:10.1007/978-3-319-99154-2_2

Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. 2023. An Algebra of Alignment for Relational Verification. *Proc. ACM Program. Lang.* POPL (2023). doi:10.1145/3571213

Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. 2017. Hypercollecting semantics and its application to static analysis of information flow. In *Symposium on Principles of Programming Languages, POPL 2017.* doi:10.1145/3009837.3009889

Ali Bajwa, Minjian Zhang, Rohit Chadha, and Mahesh Viswanathan. 2023. Stack-Aware Hyperproperties. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023.* doi:10.1007/978-3-031-30823-9_16

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Computer Security Foundations Workshop, CSFW 2004.* doi:10.1109/CSFW.2004.17

Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. 2022. Information-flow Interfaces. In *International Conference on Fundamental Approaches to Software Engineering, FASE 2022.* doi:10.1007/978-3-030-99429-7_1

Ezio Bartocci, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. 2023. Hypernode Automata. In *International Conference on Concurrency Theory, CONCUR 2023.* doi:10.4230/LIPICS.CONCUR.2023.21

Jon Barwise. 1977. An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics.*

Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. 2021. A Temporal Logic for Asynchronous Hyperproperties. In *International Conference on Computer Aided Verification, CAV 2021.* doi:10.1007/978-3-030-81685-8_33

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Symposium on Principles of Programming Languages, POPL 2004.* doi:10.1145/964001.964003

Raven Beutner. 2024. Automated Software Verification of Hyperliveness. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2024.* doi:10.1007/978-3-031-57249-4_10

Raven Beutner and Bernd Finkbeiner. 2022a. Prophecy Variables for Hyperproperty Verification. In *Computer Security Foundations Symposium, CSF 2022.* doi:10.1109/CSF54842.2022.9919658

Raven Beutner and Bernd Finkbeiner. 2022b. Software Verification of Hyperproperties Beyond k-Safety. In *International Conference on Computer Aided Verification, CAV 2022.* doi:10.1007/978-3-031-13185-1_17

Raven Beutner and Bernd Finkbeiner. 2023a. AutoHyper: Explicit-State Model Checking for HyperLTL. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023.* doi:10.1007/978-3-031-30823-9_8

Raven Beutner and Bernd Finkbeiner. 2023b. HyperATL*: A Logic for Hyperproperties in Multi-Agent Systems. *Log. Methods Comput. Sci.* (2023). doi:10.46298/LMCS-19(2:13)2023

Raven Beutner and Bernd Finkbeiner. 2024. Non-deterministic Planning for Hyperproperty Verification. In *International Conference on Automated Planning and Scheduling, ICAPS 2024.* doi:10.1609/ICAPS.V34I1.31457

Raven Beutner and Bernd Finkbeiner. 2025a. AutoHyper: leveraging Language Inclusion Checking for Hyperproperty Model-Checking. *Int. J. Softw. Tools Technol. Transf.* (2025). doi:10.1007/s10009-025-00801-5

Raven Beutner and Bernd Finkbeiner. 2025b. Multiplayer Games With Incomplete Information for Hyperproperty Verification. In *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025*. doi:10.5555/3709347.3743896

Raven Beutner and Bernd Finkbeiner. 2025c. On Hyperproperty Verification, Quantifier Alternations, and Games under Partial Information. In *Formal Methods in Computer-Aided Design, FMCAD 2025*.

Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger. 2023. Second-Order Hyperproperties. In *International Conference on Computer Aided Verification, CAV 2023*. doi:10.1007/978-3-031-37703-7_15

Raven Beutner, Tzu-Han Hsu, Borzoo Bonakdarpour, and Bernd Finkbeiner. 2024. Syntax-Guided Automated Program Repair for Hyperproperties. In *International Conference on Computer Aided Verification CAV 2024*. doi:10.1007/978-3-031-65633-0_1

Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A constraint-based approach to solving games on infinite graphs. In *Symposium on Principles of Programming Languages, POPL 2014*. doi:10.1145/2535838.2535860

Sebastian Biewer, Rayna Dimitrova, Michael Fries, Maciej Gazda, Thomas Heinze, Holger Hermanns, and Mohammad Reza Mousavi. 2022. Conformance Relations and Hyperproperties for Doping Detection in Time and Space. *Log. Methods Comput. Sci.* (2022). doi:10.46298/LMCS-18(1:14)2022

Laura Bozzelli, Bastien Maubert, and Sophie Pinchinat. 2015. Unifying Hyper and Epistemic Temporal Logics. In *International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2015*. doi:10.1007/978-3-662-46678-0_11

Laura Bozzelli, Adriano Peron, and César Sánchez. 2021. Asynchronous Extensions of HyperLTL. In *Symposium on Logic in Computer Science, LICS 2021*. doi:10.1109/LICS52264.2021.9470583

Laura Bozzelli, Adriano Peron, and César Sánchez. 2022. Expressiveness and Decidability of Temporal Logics for Asynchronous Hyperproperties. In *International Conference on Concurrency Theory, CONCUR 2022*. doi:10.4230/LIPIcs.CONCUR.2022.27

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2012. Continuity and robustness of programs. *Commun. ACM* (2012). doi:10.1145/2240236.2240262

Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. Stuttering for Free. *Proc. ACM Program. Lang.* OOPSLA (2023). doi:10.1145/3622857

Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *International Conference on Computer Aided Verification, CAV 2002*. doi:10.1007/3-540-45657-0_29

Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *International Conference on Principles of Security and Trust, POST 2014*. doi:10.1007/978-3-642-54792-8_15

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Computer Security Foundations Symposium, CSF 2008*. doi:10.1109/CSF.2008.7

Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. 2019a. The Hierarchy of Hyperlogics. In *Symposium on Logic in Computer Science, LICS 2019*. doi:10.1109/LICS.2019.8785713

Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. 2019b. Verifying Hyperliveness. In *International Conference on Computer Aided Verification, CAV 2019*. doi:10.1007/978-3-030-25540-4_7

Arthur Correnson and Bernd Finkbeiner. 2025. Coinductive Proofs for Temporal Hyperliveness. *Proc. ACM Program. Lang.* POPL (2025). doi:10.1145/3704889

Arthur Correnson, Tobias Nießen, Bernd Finkbeiner, and Georg Weissenbacher. 2024. Finding $\forall\exists$ Hyperbugs using Symbolic Execution. *Proc. ACM Program. Lang.* OOPSLA (2024). doi:10.1145/3689761

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages, POPL 1977*. doi:10.1145/512950.512973

Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* PLDI (2024). doi:10.1145/3656437

Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. 2001. Symbolic Algorithms for Infinite-State Games. In *International Conference on Concurrency Theory, CONCUR 2001*. doi:10.1007/3-540-44685-0_36

Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. 2022. RHLE: Modular Deductive Verification of Relational $\forall\exists$ Properties. In *Asian Symposium on Programming Languages and Systems, APLAS 2022*. doi:10.1007/978-3-031-21037-2_4

Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah. 2020. Probabilistic Hyperproperties of Markov Decision Processes. In *International Symposium on Automated Technology for Verification and Analysis, ATVA 2020*. doi:10.1007/978-3-030-

59152-6_27

Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proc. ACM Program. Lang.* OOPSLA (2022). doi:10.1145/3563298

Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. 2022. From Spot 2.0 to Spot 2.10: What's New?. In *International Conference on Computer Aided Verification, CAV 2022.* doi:10.1007/978-3-031-13188-2_9

Marco Eilers, Thibault Dardinier, and Peter Müller. 2023. CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity. *Proc. ACM Program. Lang.* PLDI (2023). doi:10.1145/3591289

Marco Eilers, Peter Müller, and Samuel Hitz. 2020. Modular Product Programs. *ACM Trans. Program. Lang. Syst.* (2020). doi:10.1145/3324783

Azadeh Farzan and Zachary Kincaid. 2018. Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang.* POPL (2018). doi:10.1145/3158149

Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound sequentialization for concurrent program verification. In *International Conference on Programming Language Design and Implementation, PLDI 2022.* doi:10.1145/3519939.3523727

Azadeh Farzan and Anthony Vandikas. 2020. Reductions for safety proofs. *Proc. ACM Program. Lang.* POPL (2020). doi:10.1145/3371081

Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Leander Tentrup. 2020. Realizing omega-regular Hyperproperties. In *International Conference on Computer Aided Verification, CAV 2020.* doi:10.1007/978-3-030-53291-8_4

Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL*. In *International Conference on Computer Aided Verification, CAV 2015.* doi:10.1007/978-3-319-21690-4_3

Giuseppe De Giacomo, Paolo Felli, Marco Montali, and Giuseppe Perelli. 2021. HyperLDLf: a Logic for Checking Properties of Finite Traces Process Logs. In *International Joint Conference on Artificial Intelligence, IJCAI 2021.* doi:10.24963/IJCAI.2021/256

Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman P. Amarasinghe, and Ilya Sergey. 2024. Mechanised Hypersafety Proofs about Structured Data. *Proc. ACM Program. Lang.* PLDI (2024). doi:10.1145/3656403

Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *Symposium on Security and Privacy, SP 1982.* doi:10.1109/SP.1982.10014

Ohad Goudsmid, Orna Grumberg, and Sarai Sheinvald. 2021. Compositional Model Checking for Multi-properties. In *International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2021.* doi:10.1007/978-3-030-67067-2_4

Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *International Conference on Computer Aided Verification, CAV 1997.* doi:10.1007/3-540-63166-6_10

Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Symposium on Security and Privacy, SP 2020.* doi:10.1109/SP40000.2020.00011

Jens Oliver Gutsfeld, Arne Meier, Christoph Ohrem, and Jonni Virtema. 2022. Temporal Team Semantics Revisited. In *Symposium on Logic in Computer Science, LICS 2022.* doi:10.1145/3531130.3533360

Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. 2020. Propositional Dynamic Logic for Hyperproperties. In *International Conference on Concurrency Theory, CONCUR 2020.* doi:10.4230/LIPICS.CONCUR.2020.50

Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. 2021. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.* POPL (2021). doi:10.1145/3434319

Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. 2024. Deciding Asynchronous Hyperproperties for Recursive Programs. *Proc. ACM Program. Lang.* POPL (2024). doi:10.1145/3632844

Philippe Heim and Rayna Dimitrova. 2024. Solving Infinite-State Games via Acceleration. *Proc. ACM Program. Lang.* POPL (2024). doi:10.1145/3632899

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* (1990). doi:10.1145/78969.78972

Tzu-Han Hsu, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. 2023. Bounded Model Checking for Asynchronous Hyperproperties. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023.* doi:10.1007/978-3-031-30823-9_2

Marieke Huisman, Pratik Worah, and Kim Sunesen. 2006. A Temporal Logic Characterisation of Observational Determinism. In *Computer Security Foundations Workshop, CSFW 2006).* doi:10.1109/CSFW.2006.6

Shachar Itzhaky, Sharon Shoham, and Yakir Vizel. 2024. Hyperproperty Verification as CHC Satisfiability. In *European Symposium on Programming, ESOP 2024.* doi:10.1007/978-3-031-57267-8_9

Andreas Krebs, Arne Meier, Jonni Virtema, and Martin Zimmermann. 2018. Team Semantics for the Specification and Verification of Hyperproperties. In *International Symposium on Mathematical Foundations of Computer Science, MFCS 2018.* doi:10.4230/LIPICS.MFCS.2018.10

Maurice Laveaux, Wieger Wesselink, and Tim A. C. Willemse. 2022. On-The-Fly Solving for Symbolic Parity Games. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022.* doi:10.1007/978-3-030-99527-0_8

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages, POPL 2006.* doi:10.1145/1111037.1111042

Kenji Maillard, Catalin Hritcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *Proc. ACM Program. Lang.* POPL (2020). doi:10.1145/3371072

Donald A Martin. 1975. Borel determinacy. *Annals of Mathematics* 2 (1975).

Daryl McCullough. 1988. Noninterference and the composability of security properties. In *Symposium on Security and Privacy, SP 1997.* doi:10.1109/SECPRI.1988.8110

John McLean. 1994. A general theory of composition for trace sets closed under selective interleaving functions. In *Symposium on Security and Privacy, SP 1994.* doi:10.1109/RISP.1994.296590

Robert McNaughton. 1993. Infinite Games Played on Finite Graphs. *Ann. Pure Appl. Logic* (1993). doi:10.1016/0168-0072(93)90036-D

Robin Milner. 1971. An Algebraic Definition of Simulation Between Programs. In *International Joint Conference on Artificial Intelligence, IJCAI 1971.* http://ijcai.org/Proceedings/71/Papers/044.pdf

Kedar S. Namjoshi. 1997. A Simple Characterization of Stuttering Bisimulation. In *Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 1997.* doi:10.1007/BFB0058037

Kedar S. Namjoshi and Lucas M. Tabajara. 2020. Witnessing Secure Compilation. In *International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2020.* doi:10.1007/978-3-030-39322-9_1

Nir Piterman. 2007. From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata. *Log. Methods Comput. Sci.* (2007). doi:10.2168/LMCS-3(3:5)2007

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Symposium on Foundations of Computer Science, FOCS 1977.* doi:10.1109/SFCS.1977.32

Markus N. Rabe. 2016. *A temporal logic approach to information-flow control.* Ph. D. Dissertation. Saarland University.

John H. Reif. 1984. The Complexity of Two-Player Games of Incomplete Information. *J. Comput. Syst. Sci.* (1984). doi:10.1016/0022-0000(84)90034-5

Stanly Samuel, Deepak D'Souza, and Raghavan Komondoor. 2021. GenSys: a scalable fixed-point engine for maximal controller synthesis over infinite state spaces. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021.* doi:10.1145/3468264.3473126

Anne-Kathrin Schmuck, Philippe Heim, Rayna Dimitrova, and Satya Prakash Nayak. 2024. Localized Attractor Computations for Infinite-State Games. In *International Conference on Computer Aided Verification, CAV 2024.* doi:10.1007/978-3-031-65633-0_7

Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property Directed Self Composition. In *International Conference on Computer Aided Verification, CAV 2019.* doi:10.1007/978-3-030-25540-4_9

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Conference on Programming Language Design and Implementation, PLDI 2016.* doi:10.1145/2908080.2908092

Colin Stirling. 1995. Modal and Temporal Logics for Processes. In *Logics for Concurrency - Structure versus Automata.* doi:10.1007/3-540-60915-6_5

Tachio Terauchi. 2008. A Type System for Observational Determinism. In *Computer Security Foundations Symposium, CSF 2008.* doi:10.1109/CSF.2008.9

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *International Conference on Computer Aided Verification, CAV 2021.* doi:10.1007/978-3-030-81685-8_35

Tom van Dijk. 2018. Oink: An Implementation and Evaluation of Modern Parity Game Solvers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018.* doi:10.1007/978-3-319-89960-2_16

Huanyu Wang, Domenic Forte, Mark M. Tehranipoor, and Qihang Shi. 2017. Probing Attacks on Integrated Circuits: Challenges and Research Opportunities. *IEEE Des. Test* (2017). doi:10.1109/MDAT.2017.2729398

Steve Zdancewic and Andrew C. Myers. 2003. Observational Determinism for Concurrent Program Security. In *Computer Security Foundations Workshop, CSFW 2003.* doi:10.1109/CSFW.2003.1212703

Received 2025-03-25; accepted 2025-08-12