

HYPERATL^{*}: A LOGIC FOR HYPERPROPERTIES IN MULTI-AGENT SYSTEMS

RAVEN BEUTNER  AND BERND FINKBEINER 

CISPA Helmholtz Center for Information Security, Germany

ABSTRACT. Hyperproperties are system properties that relate multiple computation paths in a system and are commonly used to, *e.g.*, define information-flow policies. In this paper, we study a novel class of hyperproperties that allow reasoning about strategic abilities in multi-agent systems. We introduce **HyperATL^{*}**, an extension of computation tree logic with path variables and strategy quantifiers. Our logic supports quantification over paths in a system – as is possible in hyperlogics such as **HyperCTL^{*}** – but resolves the paths based on the strategic choices of a coalition of agents. This allows us to capture many previously studied (strategic) security notions in a unifying hyperlogic. Moreover, we show that **HyperATL^{*}** is particularly useful for specifying asynchronous hyperproperties, *i.e.*, hyperproperties where the execution speed on the different computation paths depends on the choices of a scheduler. We show that finite-state model checking of **HyperATL^{*}** is decidable and present a model checking algorithm based on alternating automata. We establish that our algorithm is asymptotically optimal by proving matching lower bounds. We have implemented a prototype model checker for a fragment of **HyperATL^{*}** that can check various security properties in small finite-state systems.

1. INTRODUCTION

Hyperproperties [CS10] are system properties that relate multiple computation paths of a system. Such properties are of increasing importance as they can, for example, characterize the information-flow in a system. A prominent logic to express hyperproperties is **HyperLTL**, which extends linear-time temporal logic (LTL) with explicit path quantification [CFK⁺14]. **HyperLTL** can, for instance, express generalized non-interference (GNI) [McC88], stating that the high-security input of a system does not influence the observable output:

$$\forall \pi. \forall \pi'. \exists \pi''. \Box \left(\bigwedge_{a \in H} a_\pi \leftrightarrow a_{\pi''} \right) \wedge \Box \left(\bigwedge_{a \in O} a_{\pi'} \leftrightarrow a_{\pi''} \right) \quad (GNI)$$

Key words and phrases: hyperproperties, multi-agent systems, alternating-time temporal logic, **HyperATL^{*}**, information-flow control, asynchronous hyperproperties, model checking, non-interference, **HyperLTL**, **HyperCTL^{*}**.

This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center “Foundations of Perspicuous Software Systems” (TRR 248, 389792660) and by the ERC Grants OSARES (No. 683300) and HYPER (No. 101055412). R. Beutner carried out this work as a member of the Saarbrücken Graduate School of Computer Science.

Here H is a set of high-security input propositions and O a set of outputs (for simplicity, we assume that there are no low-security inputs). In our model, the system thus generates a set of computations paths over $H \cup O$, where each path corresponds to a possible input-output interaction with the system. The *GNI* formula then states that for any pair of paths π, π' there exists a third path π'' that agrees with the high-security inputs of π and with the outputs of π' . The existence of π'' guarantees that any observation on the outputs is compatible with every possible sequence of high-security inputs; non-determinism is the sole explanation for the output.

Existing hyperlogics (like **HyperLTL**) consider a system as a set of paths and quantify (universally or existentially) over those paths. In this paper, we introduce a novel class of hyperproperties that reason about *strategic behavior* in a multi-agent system where the paths of the system are outcomes of games played on a game structure. We introduce **HyperATL***, a temporal logic to express hyperproperties in multi-agent systems. Our logic builds on the foundation laid by alternating-time temporal logics (**ATL***) [AHK02].¹ While strategy quantifiers in **ATL*** can be nested (similar to **CTL***), the logic is unable to express hyperproperties, as the scope of each quantifier is limited to the current path.

In **HyperATL***, we combine quantification over strategic behavior with the ability to express hyperproperties. Syntactically, our logic combines the strategic quantifier of **ATL*** but binds the outcome to a path variable:² The **HyperATL*** formula $\langle\langle A \rangle\rangle \pi. \varphi$ specifies that the agents in A have a strategy such that all outcomes under that strategy, when bound to the path variable π , satisfy φ . Similar to **HyperLTL**, quantification is resolved incrementally. For example, $\langle\langle A \rangle\rangle \pi. \langle\langle A' \rangle\rangle \pi'. \varphi$ requires the existence of strategy for the agents in A such that for all possible outcomes under that strategy, when bound to π , the agents in A' have a strategy such that all possible outcomes, when bounds to π' , satisfy φ . In particular, the quantification over strategies for A' takes

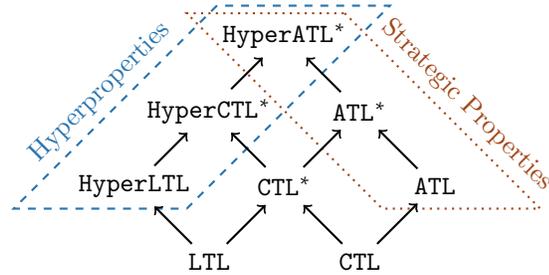


FIGURE 1. Hierarchy of expressiveness of temporal logics. An arrow $A \rightarrow B$ indicates that A is a (syntactic) fragment of B . Logics in the blue, dashed area can express hyperproperties. Logics in the red, dotted area can express strategic properties in multi-agent systems. Logics that are interpreted on multi-agent systems (**ATL**, **ATL***, and **HyperATL***) can also be applied to transition systems (the standard model for the remaining logics) by interpreting transition systems as 1-agent systems (see Remark 2.1); the reverse does not hold, *i.e.*, logics that are interpreted on transitions systems cannot reason about strategic abilities in multi-agent systems.

¹**ATL*** is a temporal logic that extends **CTL*** by offering selective quantification over paths that are possible outcomes of games [AHK02]. The **ATL*** quantifier $\langle\langle A \rangle\rangle \varphi$ states that the players in A have a joint strategy such that every outcome under that strategy satisfies φ .

²Similar to logics such as **HyperLTL**, we use path variables as a syntactic tool to refer to paths that are bound by outer quantifiers. For example, in the *GNI* formula, we use paths variables π, π', π'' and compare the paths bound to these variables in the body of the formula.

place after π is fixed.³ We endow our logic with an explicit construct to resolve multiple games simultaneously (syntactically, we surround quantifiers by $[\cdot]$ brackets). The formula $[\langle\langle A \rangle\rangle\pi. \langle\langle A' \rangle\rangle\pi'] \varphi$ requires winning strategies for the agents in A (for the first copy) and for A' (for the second copy) in a game that progresses simultaneously, *i.e.*, the players can observe the current state of both copies.

The resulting logic is very expressive and subsumes both the existing hyperlogic **HyperCTL*** [CFK⁺14] (the branching-time extension of **HyperLTL**) and the alternating-time logic **ATL*** [AHK02]. The resulting expressiveness hierarchy is depicted in Figure 1.

Reasoning about strategic hyperproperties in multi-agent systems is useful in various settings, including information-flow control and asynchronous hyperproperties. Consider the following two examples that demonstrate how we can use **HyperATL*** to express such strategic hyperproperties.

Application 1: Information-flow Control. We consider a strategic information-flow control property. Imagine a system where the non-determinism arises from a scheduling decision between two subprograms P_1 and P_2 . Each subprogram reads the next high-security input h of the system. Suppose that P_1 assumes that h is even and otherwise leaks information, while P_2 assumes that h is odd and otherwise leaks information. We can check *GNI* on the resulting system. In **HyperLTL**, quantification is resolved incrementally, so the witness path π'' is chosen after π and π' are already fixed. In particular, all future high-security input h are already determined, so a leakage disproving path π'' can be constructed (by always scheduling the copy that does not leak information on the *next* input); the system satisfies *GNI*. By contrast, an *actual* scheduler, who determines which subprogram handles the next input, cannot avoid information leakage. **HyperATL*** can express a stricter information-flow policy. As a first step, we consider the system as a game structure with two players. Player ξ_H chooses (in each step) the values for the high-security inputs, and player ξ_N resolves the remaining non-determinism of the system (*i.e.*, the nondeterminism not caused by input selection). We give a concrete semantics into such a game structure in Section 8.1. Consider the following **HyperATL*** specification:

$$\forall \pi. \langle\langle \xi_N \rangle\rangle \pi'. \Box \left(\bigwedge_{a \in O} a_\pi \leftrightarrow a_{\pi'} \right)$$

This formula requires that for every possible path π , the non-determinism player ξ_N has a *strategy* to produce identical outputs on π' against all possible moves by ξ_H . As we consider the system as a game structure, any strategy for ξ_N does not know which moves ξ_H will play in the future. Any possible output of the system is thus achievable by a strategy only knowing the finite input history but oblivious to the future high-security inputs. The system sketched above does not satisfy this property.

A particular strength of this formulation is that we can encode additional requirements on the strategy for the scheduler. For example, if the internal non-determinism arises from the scheduling decisions between multiple components, we can, in addition, require fairness of the scheduling strategy. ◀

³This incremental elimination of quantification ensures that **HyperATL*** is a proper extension of **HyperCTL*** but is also crucial for decidable model checking. In fact, model checking for any logic that could express the existence of a strategy such that the set of outcomes under that strategy satisfies a hyperproperty, would already subsume the realizability problem for hyperproperties, which is known to be undecidable already for very simple fragments [FHL⁺18].

Application 2: Asynchronous Hyperproperties.

Most existing hyperlogics traverse the paths of a system synchronously. However, in many applications (for example, when reasoning about software), we require an asynchronous traversal to, *e.g.*, account for the unknown speed of execution of software that runs on some unknown platform or to abstract away from intermediate (non-observable) program steps. Strategic hyperproperties enable reasoning over asynchronous hyperproperties by considering the execution speed of a system as being controlled by a dedicated scheduling player, which we add via a system transformation. Direct reasoning about asynchronicity is then replaced by reasoning about the strategic abilities of the scheduling player.

As an example, consider the program in Figure 2. It continuously reads a high-security input h and, depending on h , flips the output o either directly or via a temporary variable. Consider the **HyperLTL** specification $\forall \pi. \forall \pi'. \Box(o_\pi \leftrightarrow o_{\pi'})$. It expresses *observational-determinism* (OD), *i.e.*, it states that the output should be identical across all paths [HWS06]. In Figure 2, the value of o is flipped in each loop iteration, but the exact timepoint at which the flip occurs depends on the high-security input; the program does not satisfy OD in the synchronous **HyperLTL** semantics. However, when executing the program, we might assume that an observer cannot detect this small timing difference and, instead, only observes the value of o whenever it is changed. We thus require a property that allows the paths of the program to be executed at different speeds to realign the output. To reason about this in **HyperATL***, we extend the system with a scheduling agent *sched* that can stutter the system. That is, *sched* can, in each step, decide if the system makes a step or remains in its current state (we give a concrete construction for including *sched* in Section 5). On the resulting multi-agent system (which now includes agent *sched*) we check the following **HyperATL*** property:

$$[\langle\langle sched \rangle\rangle \pi. \langle\langle sched \rangle\rangle \pi'] \text{fair}_\pi \wedge \text{fair}_{\pi'} \wedge \Box(o_\pi \leftrightarrow o_{\pi'})$$

This formula requires that the scheduler has a strategy to align any two program paths such that the output agrees. The additional requirement $\text{fair}_\pi \wedge \text{fair}_{\pi'}$ ensures that both copies are scheduled infinitely many times (see Section 5 for details). By surrounding the quantifier with $[\cdot]$, both paths are resolved simultaneously (instead of incrementally), so the strategies in both copies can collaborate. The program in Figure 2 (with an added asynchronous scheduler) satisfies this **HyperATL*** property. Any two paths in the original system can thus be aligned (by stuttering for any finite number of steps) such that the output agrees globally.

This general style of asynchronous reasoning turns out to be remarkably effective: Verification is possible (decidable) for the full logic, and the approach subsumes the largest known decidable fragment of a recent asynchronous extension of **HyperLTL** [BCB⁺21]. ◀

```

o ← true
while(true)
  h ← ReadH
  if h then
    o ← ¬o
  else
    temp ← ¬o
    o ← temp

```

FIGURE 2. Example program that violates (synchronous) observational determinism.

Model Checking. We show that model checking of **HyperATL*** on concurrent game structures is decidable and present an automata-based model checking algorithm. Our algorithm incrementally reduces model checking to the emptiness of an automaton. By using alternating automata, we encode the strategic behavior of a game structure within the transition function of the automaton. To characterize its complexity, we partition **HyperATL*** formulas based

on the number of complex quantifiers (*i.e.*, quantifiers where the set of agents is non-trivial) and simple quantifiers (*i.e.*, quantifiers that reason about all possible paths, irrespective of the strategic behavior in the system). For each fragment, we derive upper bounds on the model checking complexity, both in the size of the system and specification. Different from **HyperLTL** – where each alternation results in an exponential blowup [FRS15, Rab16, BF23a] – the strategic quantification in **HyperATL^{*}** results in a *double* exponential blowup with each complex quantifier. Using a novel counter construction, we prove matching lower bounds on the **HyperATL^{*}** model checking problem (in the size of both specification and system).

Prototype Model Checker. On the practical side, we present `hyperatlmc`, a prototype model checker for a fragment of **HyperATL^{*}**. The fragment supported by our tool does, in particular, include all alternation-free **HyperLTL** formulas [FRS15], the model checking approach from [CFST19], and the formulas arising when expressing asynchronous hyperproperties in **HyperATL^{*}**.

Contributions. In summary, our contributions include the following:

- We introduce **HyperATL^{*}**, a novel logic to express strategic hyperproperties in multi-agent systems. We demonstrate that **HyperATL^{*}** can express many existing information-flow policies and offers a natural formalism to express asynchronous hyperproperties (subsuming the largest known decidable fragment of the logic presented in [BCB⁺21]).
- We give an automata-based model checking algorithm for **HyperATL^{*}** and analyze its complexity (both in system and specification size) based on the number and type of quantifiers.
- We prove matching lower bounds on the **HyperATL^{*}** model checking problem via a novel counter construction.
- We present `hyperatlmc`, a prototype-model checker for a fragment of **HyperATL^{*}**, and use it to verify information-flow policies and asynchronous hyperproperties in small systems.

This paper is an extended version of a preliminary conference version [BF21]. Compared to the conference paper, this version contains detailed and streamlined proofs of the theoretical results. Moreover, we extend our earlier complexity bounds by also analyzing **HyperATL^{*}** model-checking in the size of the system (in [BF21] we only consider the complexity in the size of the specification) and derive uniform lower and upper bounds that are stated purely in the number and type of the quantifiers (without side conditions needed in [BF21]).

Structure. The remainder of this paper is structured as follows. In Section 2, we introduce basic preliminaries and in Section 3 we develop **HyperATL^{*}**. Afterward, in Sections 4 and 5, we present examples (ranging from information-flow policies to asynchronous hyperproperties) expressible in **HyperATL^{*}** and connect to existing asynchronous hyperlogics. We discuss model checking on finite-state game structures in Section 6. In Section 7, we show matching lower bounds on the model checking problem. Finally, in Section 8, we report on our prototype model checker and discuss related work in Section 9.

2. PRELIMINARIES

In this section, we introduce basic preliminaries on transition systems, game structures and alternating automata. For a set X , we write X^* for the set of finite sequences over X , X^+ for the set of non-empty finite sequences, and X^ω for the set of infinite sequences. For an

infinite sequence $u \in X^\omega$ and $i \in \mathbb{N}$, we write $u(i) \in X$ for the i th element (starting at the 0th) and $u[i, \infty] \in X^\omega$ for the infinite-suffix starting at position i . For $u_1, \dots, u_n \in X^\omega$, we define $\otimes(u_1, \dots, u_n) \in (X^n)^\omega$ as the pointwise product, *i.e.*, $\otimes(u_1, \dots, u_n)(i) := (u_1(i), \dots, u_n(i))$. In case of only two sequences u_1, u_2 , we write $u_1 \otimes u_2$ instead of $\otimes(u_1, u_2)$. The pointwise product extends to finite sequences of the same length. We fix a finite set of atomic propositions AP and define $\Sigma := 2^{AP}$.

Transition Systems. A *transition system* is a tuple $\mathcal{T} = (S, s_0, \delta, L)$ where S is a finite set of states, $s_0 \in S$ is an initial state, $\delta \subseteq S \times S$ is a transition relation, and $L : S \rightarrow \Sigma$ is a labeling function. We assume that for every $s \in S$ there is at least one s' such that $(s, s') \in \delta$. A path in \mathcal{T} is an infinite sequence $p \in S^\omega$ such that $p(0) = s_0$ and for every $i \in \mathbb{N}$, $(p(i), p(i+1)) \in \delta$.

Concurrent Game Structures. As the basic model of multi-agent systems we use game structures. A *concurrent game structure* (CGS) [AHK02] is an extension of a transition system in which the transition relation is composed of the moves of individual agents (also called players). Formally, a CGS is a tuple $\mathcal{G} = (S, s_0, \Xi, \mathcal{M}, \delta, L)$. The finite set of states S , the initial state $s_0 \in S$, and the labeling $L : S \rightarrow \Sigma$ are as in a transition system. Additionally, Ξ is a finite and non-empty set of agents (or players), \mathcal{M} is a finite and non-empty set of moves, and $\delta : S \times (\Xi \rightarrow \mathcal{M}) \rightarrow S$ is a transition function. We call a mapping $\sigma : \Xi \rightarrow \mathcal{M}$ a *global* move vector. Given a state and global move vector, the transition function δ determines a unique successor state. For a set of agents $A \subseteq \Xi$ we call a function $\sigma : A \rightarrow \mathcal{M}$ a *partial* move vector. For disjoint sets of agents $A_1, A_2 \subseteq \Xi$ and partial move vectors $\sigma_i : A_i \rightarrow \mathcal{M}$ for $i \in \{1, 2\}$ we define $\sigma_1 + \sigma_2 : A_1 \cup A_2 \rightarrow \mathcal{M}$ as the move vector obtained as the combination of the individual choices. For $\sigma : A \rightarrow \mathcal{M}$ and $A' \subseteq A$, we define $\sigma|_{A'} : A' \rightarrow \mathcal{M}$ by restricting the domain of σ to A' .

Remark 2.1. We can naturally interpret a transition systems as a 1-player CGS in which the move of the unique player determines the successor state of the system. Any temporal logic that specifies properties on game structures is thus also applicable in transition systems. \triangleleft

Multi Stage Concurrent Game Structures. In a concurrent game structure (as the name suggests), all agents choose their next move concurrently, *i.e.*, without knowing what moves the other players have chosen. We introduce the concept of a *multi-stage game structure* (MSCGS), in which the move selection proceeds in stages and agents can base their decision on the already fixed moves of (some of the) other agents. This is particularly useful when we, *e.g.*, want to base a scheduling decision on the moves selected by the other agents. Formally, a MSCGS is a CGS equipped with a function $d : \Xi \rightarrow \mathbb{N}$, that orders the agents according to informedness. Whenever $d(\xi_1) < d(\xi_2)$, ξ_2 can base its next move on the move selected by ξ_1 . A CGS thus naturally corresponds to a MSCGS with $d = \mathbf{0}$, where $\mathbf{0}$ is the constant 0 function.

Strategies in Game Structures. A strategy in a game structure is a function that maps finite histories of plays in the game to a move in \mathcal{M} . As the plays in an MSCGS progress in stages, each decision is based on the past sequence of states *and* the fixed moves of all agents in previous stages. Formally, a strategy for an agent $\xi \in \Xi$ is a function

$$f_\xi : S^+ \times (\{\xi' \mid d(\xi') < d(\xi)\} \rightarrow \mathcal{M}) \rightarrow \mathcal{M}.$$

Note that in case where $d = \mathbf{0}$, a strategy can be seen as a function $S^+ \rightarrow \mathcal{M}$.

Definition 2.2. Given a set of agents A , a set of strategies $F_A = \{f_\xi \mid \xi \in A\}$, and a state $s \in S$, we define $out(\mathcal{G}, s, F_A) \subseteq S^\omega$ as the set of all runs $u \in S^\omega$ such that 1) $u(0) = s$, and 2) for every $i \in \mathbb{N}$ there exists a global move vector $\sigma : \Xi \rightarrow \mathcal{M}$ with $\delta(u(i), \sigma) = u(i+1)$ and for all $\xi \in A$ we have $\sigma(\xi) = f_\xi(u[0, i], \sigma_{|\{\xi' \mid d(\xi') < d(\xi)\}})$. \triangleleft

Alternating Automata. For a set X , we write $\mathbb{B}^+(X)$ for the set of positive boolean formulas over X with the standard propositional semantics. Given $\Psi \in \mathbb{B}^+(X)$ and $B \subseteq X$ we write $B \models \Psi$ if the assignment obtained from B by mapping all $x \in B$ to true and all $x \notin B$ to false satisfies Ψ . An *alternating parity automaton* (APA) is a tuple $\mathcal{A} = (Q, q_0, \Sigma, \rho, c)$ where Q is a finite set of states, $q_0 \in Q$ is an initial state, Σ is a finite alphabet, $\rho : Q \times \Sigma \rightarrow \mathbb{B}^+(Q)$ is a transition function, and $c : Q \rightarrow \mathbb{N}$ is a coloring of states. A tree is a set $T \subseteq \mathbb{N}^*$ that is prefixed closed, *i.e.*, $\tau \cdot n \in T$ implies $\tau \in T$. We refer to elements in $\tau \in T$ as nodes and denote with $|\tau|$ the length of τ (or equivalently the depth of the node). For a node $\tau \in T$, we define $children(\tau)$ as the set of immediate children of τ , *i.e.*, $children(\tau) := \{\tau \cdot n \in T \mid n \in \mathbb{N}\}$. An X -labeled tree is a pair (T, r) where T is a tree and $r : T \rightarrow X$ labels nodes with an element in X . A run of an APA $\mathcal{A} = (Q, q_0, \Sigma, \rho, c)$ on a word $u \in \Sigma^\omega$ is a Q -labeled tree (T, r) such that 1) $r(\epsilon) = q_0$, and 2) for all $\tau \in T$, $\{r(\tau') \mid \tau' \in children(\tau)\} \models \rho(r(\tau), u(|\tau|))$. A run (T, r) is accepting if, for every infinite path in T , the minimal color that occurs infinitely many times (as given by c) is even. We denote with $\mathcal{L}(\mathcal{A})$ the set of words for which \mathcal{A} has an accepting run. We call an alternating automaton \mathcal{A} non-deterministic (resp. universal) if the codomain of the transition function ρ consists of disjunctions (resp. conjunction) of states. If the codomain of ρ consists of atomic formulas (*i.e.*, formulas without boolean connectives that only consist of a single positive state atom), we call \mathcal{A} deterministic.⁴ Alternating, non-deterministic, universal, and deterministic parity automata all recognize the same class of languages (namely ω -regular ones) although they can be (double) exponentially more succinct.

Theorem 2.3 [MH84, DH94]. *For every alternating parity automaton \mathcal{A} with n states, there exists a non-deterministic parity automaton \mathcal{A}' with $2^{\mathcal{O}(n \log n)}$ states that accepts the same language. For every non-deterministic or universal parity automaton \mathcal{A} with n states, there exists a deterministic parity automaton \mathcal{A}' with $2^{\mathcal{O}(n \log n)}$ states that accepts the same language.*

Theorem 2.4. *For every alternating parity automaton \mathcal{A} with n states, there exists an alternating parity automaton $\bar{\mathcal{A}}$ with $\mathcal{O}(n)$ states that accepts the complemented language. If \mathcal{A} is non-deterministic (resp. universal), $\bar{\mathcal{A}}$ is universal (resp. non-deterministic).*

3. HYPERATL^{*}

In this section, we introduce HyperATL^{*}. Our logic extends CTL^{*} [EH86] by introducing path variables (similar to HyperCTL^{*} [CFK⁺14]) and strategic quantification (similar to ATL^{*} [AHK02]). Assume a fixed set of agents Ξ and let \mathcal{V} be a finite set of path variables. HyperATL^{*} formulas are generated by the following grammar

$$\varphi, \psi := \langle\langle A \rangle\rangle \pi. \varphi \mid a_\pi \mid \varphi \wedge \psi \mid \neg \varphi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \psi$$

⁴In a non-deterministic or universal automaton, we interpret ρ as a function $Q \times \Sigma \rightarrow 2^Q$. In a deterministic automaton, we interpret ρ as a function $Q \times \Sigma \rightarrow Q$.

where $\pi \in \mathcal{V}$ is a path variable, $a \in AP$ an atomic proposition, and $A \subseteq \Xi$ a set of agents. Quantification of the form $\langle\langle A \rangle\rangle\pi.\varphi$ binds a path to path variable π , and a_π refers to the truth value of a on the path bound to variable π . A formula is closed if all sub-formulas a_π occur in the scope of a quantifier that binds π ; throughout the paper we assume all formulas to be closed. We use the usual derived boolean connectives $\vee, \rightarrow, \leftrightarrow$, the boolean constants true (\top) and false (\perp), and temporal operators *eventually* ($\Diamond\varphi := \top\mathcal{U}\varphi$), *globally* ($\Box\varphi := \neg\Diamond\neg\varphi$), and *weak until* ($\varphi\mathcal{W}\psi := (\varphi\mathcal{U}\psi) \vee \Box\varphi$).

The strategic quantifier $\langle\langle A \rangle\rangle\pi.\varphi$ postulates that the agents in A have a joint strategy such that every outcome under that strategy, when bound to path variable π , satisfies φ . Trivial agent sets, *i.e.*, $A = \emptyset$ or $A = \Xi$, correspond to classical existential or universal quantification. We therefore write $\forall\pi.\varphi$ instead of $\langle\langle \emptyset \rangle\rangle\pi.\varphi$ and $\exists\pi.\varphi$ instead of $\langle\langle \Xi \rangle\rangle\pi.\varphi$. For a single agent $\xi \in \Xi$, we sometimes write $\langle\langle \xi \rangle\rangle\pi.\varphi$ instead of $\langle\langle \{\xi\} \rangle\rangle\pi.\varphi$. We define $\llbracket A \rrbracket\pi.\varphi := \neg\langle\langle A \rangle\rangle\pi.\neg\varphi$ which states that the agents in A have *no* strategy such that every outcome, when bound to π , avoids φ (see [AHK02]). We call a quantifier $\langle\langle A \rangle\rangle\pi$ *simple* if the agent-set A is trivial (\emptyset or Ξ) and otherwise *complex*.

A **HyperATL*** formula is *linear* if it consists of an initial quantifier prefix followed by a quantifier-free formula, *i.e.*, has the form $\mathbb{Q}_1\pi_1 \dots \mathbb{Q}_n\pi_n.\psi$ where ψ is quantifier-free and each \mathbb{Q}_i is either $\langle\langle A \rangle\rangle$ or $\llbracket A \rrbracket$ for some A . **Linear-HyperATL*** is thus the syntactic subfragment of **HyperATL*** that is analogous to the definition of **HyperLTL** as a syntactic fragment of **HyperCTL*** [CFK⁺14].

Semantics. The semantics of **HyperATL*** is defined with respect to a game structure $\mathcal{G} = (S, s_0, \Xi, \mathcal{M}, \delta, d, L)$ and a path assignment Π , which is a partial mapping $\Pi : \mathcal{V} \rightarrow S^\omega$. For $\pi \in \mathcal{V}$ and path $p \in S^\omega$ we write $\Pi[\pi \mapsto p]$ for the assignment obtained by updating the value of π to p . We write $\Pi[i, \infty]$ to denote the path assignment defined by $\Pi[i, \infty](\pi) := \Pi(\pi)[i, \infty]$.

$\Pi \models_{\mathcal{G}} a_\pi$	iff	$a \in L(\Pi(\pi)(0))$
$\Pi \models_{\mathcal{G}} \neg\varphi$	iff	$\Pi \not\models_{\mathcal{G}} \varphi$
$\Pi \models_{\mathcal{G}} \varphi \wedge \psi$	iff	$\Pi \models_{\mathcal{G}} \varphi$ and $\Pi \models_{\mathcal{G}} \psi$
$\Pi \models_{\mathcal{G}} \bigcirc\varphi$	iff	$\Pi[1, \infty] \models_{\mathcal{G}} \varphi$
$\Pi \models_{\mathcal{G}} \varphi\mathcal{U}\psi$	iff	$\exists i \geq 0. \Pi[i, \infty] \models_{\mathcal{G}} \psi$ and $\forall 0 \leq j < i. \Pi[j, \infty] \models_{\mathcal{G}} \varphi$
$\Pi \models_{\mathcal{G}} \langle\langle A \rangle\rangle\pi.\varphi$	iff	$\exists F_A. \forall p \in \text{out}(\mathcal{G}, \Pi(\epsilon)(0), F_A). \Pi[\pi \mapsto p] \models_{\mathcal{G}} \varphi$

Here $\Pi(\epsilon)$ refers to the path that was last added to the assignment (similar to the **HyperCTL*** semantics [CFK⁺14]).⁵ If Π is the empty assignment, we define $\Pi(\epsilon)(0)$ as the initial state s_0 of \mathcal{G} . We say that \mathcal{G} satisfies φ , written $\mathcal{G} \models \varphi$, if $\emptyset \models_{\mathcal{G}} \varphi$ where \emptyset is the empty path assignment.

Remark 3.1. Note that the games used to produce paths in a strategy quantification $\langle\langle A \rangle\rangle\pi.\varphi$ are *local*, *i.e.*, the outcome of the game is fixed (and bound to π) before φ is evaluated further. The strategy for agents in A is quantified after the outer paths (those bound to the path variables that are free in $\langle\langle A \rangle\rangle\pi.\varphi$) are already fixed. For example, in a formula of the form $\forall\pi.\langle\langle A \rangle\rangle\pi'.\varphi$ the agents in A know the already fixed path bound to π but behave as a strategy w.r.t. π' . In particular, any formula using only simple quantification

⁵If we assume the path variables quantified in a formula are distinct (which we can always ensure by α -renaming), we can view a path assignment Π as a finite list of pairs in $\mathcal{V} \times S^\omega$, interpret $\Pi[\pi \mapsto p]$ as appending the pair (π, p) to the list, and interpret $\Pi(\epsilon)$ as the path of the last pair in the list.

(i.e., only \forall and \exists quantifiers) corresponds directly to the (syntactically identical) **HyperCTL*** property. \triangleleft

Proposition 3.2. *HyperATL* subsumes ATL*. When interpreting transitions systems as 1-player CGSs (cf. Rem. 2.1), HyperATL* subsumes HyperCTL* (and thus HyperLTL). The resulting hierarchy is depicted in Figure 1.*

Extension 1: Extended Path Quantification. Oftentimes, it is convenient to compare different game structures with respect to a hyperproperty. For *linear* **HyperATL*** properties, we consider formulas with extended path quantification. We write $\langle\langle A \rangle\rangle_{\mathcal{G}} \pi. \varphi$ to quantify path π via a game played in \mathcal{G} . For example, $\forall_{\mathcal{G}} \pi. \langle\langle A \rangle\rangle_{\mathcal{G}'} \pi'. \Box(o_{\pi} \leftrightarrow o_{\pi'})$ states that for each path π in \mathcal{G} the agents in A have a strategy in \mathcal{G}' that produces only paths π' which agree with π on o (where o is a shared proposition between \mathcal{G} and \mathcal{G}').⁶

Extension 2: Parallel Composition. We extend **HyperATL*** with a syntactic construct that allows multiple paths to be resolved in a single bigger game, where individual copies of the system progress in parallel. Consider the following modification to the **HyperATL*** syntax, where $k \geq 1$:

$$\varphi, \psi := [\langle\langle A_1 \rangle\rangle \pi_1 \dots \langle\langle A_k \rangle\rangle \pi_k] \varphi \mid a_{\pi} \mid \neg \varphi \mid \varphi \wedge \psi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \psi$$

When surrounding strategy quantifiers by $[\cdot]$, the resulting paths are the outcome of a game played on a bigger, parallel game of the structure. Consequently, the agents in each copy can base their decisions not only on the current state of their copy but on the combined state of all k copies (which allows for a coordinated behavior among the copies). For a player ξ and CGS $\mathcal{G} = (S, s_0, \Xi, \mathcal{M}, \delta, L)$, a *k-fold strategy* for ξ is a function $f_{\xi} : (S^k)^+ \rightarrow \mathcal{M}$.

Definition 3.3. For a system \mathcal{G} , sets of k -fold strategies F_{A_1}, \dots, F_{A_k} and states s_1, \dots, s_k , we define $out(\mathcal{G}, (s_1, \dots, s_k), F_{A_1}, \dots, F_{A_k})$ as all plays $u \in (S^k)^{\omega}$ such that 1) $u(0) = (s_1, \dots, s_k)$, and 2) for every $i \in \mathbb{N}$ there exist global move vectors $\sigma_1, \dots, \sigma_k$ such that $u(i+1) = (\delta(t_1, \sigma_1), \dots, \delta(t_k, \sigma_k))$ where $u(i) = (t_1, \dots, t_k)$ and for every $j \in \{1, \dots, k\}$, agent $\xi \in A_j$ and strategy $f_{\xi} \in F_{A_j}$, it holds that $\sigma_j(\xi) = f_{\xi}(u[0, i])$. \triangleleft

The definition of k -fold strategies and $out(\mathcal{G}, (s_1, \dots, s_k), F_{A_1}, \dots, F_{A_k})$ extends naturally if we consider MSCGSs instead of CGSs. We extend our semantics by the following judgment:

$$\Pi \models_{\mathcal{G}} [\langle\langle A_1 \rangle\rangle \pi_1 \dots \langle\langle A_k \rangle\rangle \pi_k] \varphi \quad \text{iff} \quad \exists F_{A_1}, \dots, F_{A_k}.$$

$$\forall (p_1, \dots, p_k) \in out(\mathcal{G}, (\Pi(\epsilon)(0), \dots, \Pi(\epsilon)(0)), F_{A_1}, \dots, F_{A_k}). \Pi[\pi_1 \mapsto p_1] \dots [\pi_k \mapsto p_k] \models_{\mathcal{G}} \varphi$$

Here we consider $out(\mathcal{G}, (s_1, \dots, s_k), F_{A_1}, \dots, F_{A_k})$ as a subset of $(S^{\omega})^k$ instead of $(S^k)^{\omega}$ using the natural correspondence (given by \otimes^{-1}). Note that $[\langle\langle A \rangle\rangle \pi] \varphi$ is equivalent to $\langle\langle A \rangle\rangle \pi. \varphi$.

⁶Formally, we change the syntax of quantification from $\langle\langle A \rangle\rangle \pi. \varphi$ to $\langle\langle A \rangle\rangle_{\nu} \pi. \varphi$ where ν is a *system variable*. We evaluate the resulting formula no longer on a single system \mathcal{G} but on a mapping N that maps system variables to game structures. Each quantifier $\langle\langle A \rangle\rangle_{\nu} \pi. \varphi$ is then resolved on system $N(\nu)$. The interesting case in the semantics thus becomes

$$\Pi \models_N \langle\langle A \rangle\rangle_{\nu} \pi. \varphi \quad \text{iff} \quad \exists F_A. \forall p \in out(N(\nu), N(\nu)_0, F_A). \Pi[\pi \mapsto p] \models_N \varphi.$$

Here, $N(\nu)_0$ is the initial state in game structure $N(\nu)$, and F_A ranges over strategies for the agents in A in game structure $N(\nu)$. Note that this is only possible for linear formulas, as each play starts in the initial state of the game structure, irrespective of the current path assignment. See [Rab16, §5.4] for details on the extended path quantification in the context of **HyperCTL***.

4. STRATEGIC HYPERPROPERTIES AND INFORMATION-FLOW CONTROL

Before discussing the automated verification of HyperATL^* properties, we consider example properties expressed in HyperATL^* . We organize our examples into two categories. We begin with examples from information-flow control and highlight the correspondence with existing properties and security paradigms (this is done in this section). Afterward (in Section 5), we show that strategic hyperproperties are naturally suited to express asynchronous hyperproperties.

Remark 4.1. In our discussion of information-flow policies, we focus on game structures that result from reactive systems. Let H, L , and O be pairwise disjoint sets of atomic propositions denoting high-security inputs, low-security inputs, and outputs, respectively. We consider a system as a 3-player game structure comprising agents ξ_N, ξ_H , and ξ_L responsible for resolving non-determinism, selecting high-security, and selecting low-security inputs, respectively. In particular, the move from ξ_H (resp. ξ_L) determines the values of the propositions in H (resp. L) in the next step. Agent ξ_N resolves the remaining non-determinism in the system. We call a CGS of the above form a *progCGS* (program-CGS). We will see a concrete transformation of programs into progCGSs in Section 8.1. For now, we rely on the reader's intuition. We call a progCGS *input-total* if, in each step, ξ_H and ξ_L can choose *all* possible valuations for the input propositions in H and L , respectively; we assume all progCGSs in this section to be input total. \triangleleft

4.1. Strategic Non-Interference. In the introduction, we already saw that, in some cases, generalized non-interference [McC88] is a too relaxed notion of security, as the witness path is fixed knowing the entire future input-output behavior. Recall the definition of GNI (compared to the definition in the introduction, we now also support low-security inputs):

$$\forall \pi. \forall \pi'. \exists \pi''. \Box \left(\bigwedge_{a \in H} a_\pi \leftrightarrow a_{\pi''} \right) \wedge \Box \left(\bigwedge_{a \in L \cup O} a_{\pi'} \leftrightarrow a_{\pi''} \right) \quad (\text{GNI})$$

In HyperATL^* , we express

$$\forall \pi. \langle\langle \{\xi_N, \xi_L\} \rangle\rangle \pi'. \Box \left(\bigwedge_{a \in L \cup O} a_\pi \leftrightarrow a_{\pi'} \right). \quad (\text{stratNI})$$

In this formula, a strategy for the ξ_N and ξ_L should construct a path π' that agrees with the low-security inputs and outputs of π . Note that ξ_L only determines the value of the propositions in L , so any winning strategy for ξ_L is “deterministic” in the sense that it needs to copy the low-security inputs from π . Agent ξ_N needs to resolve the non-determinism but does not know the future high-security inputs on π' (as those are chosen by ξ_H). If there exists a winning *strategy* for ξ_N that avoids information leakage, there also exists a path (in the sense of *GNI*) that disproves leakage.

Lemma 4.2. *For any progCGS \mathcal{G} , if $\mathcal{G} \models \text{stratNI}$ then $\mathcal{G} \models \text{GNI}$.*

Proof. We show the contraposition and assume that $\mathcal{G} \not\models \text{GNI}$. There thus exists paths p_H and $p_{L,O}$ such that no path in \mathcal{G} agrees with the high inputs of p_H and low-security inputs and outputs of $p_{L,O}$. We show that $\mathcal{G} \not\models \text{stratNI}$. For the universally quantified path bound to π we choose $p_{L,O}$ and argue that ξ_N, ξ_L have no winning strategy to construct π' . A spoiling strategy for ξ_H is the one that, in each step, chooses the high-security inputs in accordance with p_H (which is possible as the system is input-total). A winning strategy for

ξ_N and ξ_L would then need to construct a path that agrees with p_H on H and with $p_{L,O}$ on $L \cup O$ which, by assumption, does not exist. \square

4.2. Parallel Composition and $\forall\exists$ -verification. Our next examples make use of the parallel composition offered by quantification of the form $[\langle\langle A_1 \rangle\rangle\pi_1 \dots \langle\langle A_k \rangle\rangle\pi_k]$. We consider the model checking algorithm for $\forall^*\exists^*$ -HyperLTL formulas introduced by Coenen et al. [CFST19]. The idea is to consider the verification of HyperLTL formula $\forall\pi. \exists\pi'. \varphi$ as a game between the \forall -player and \exists -player. The \forall -player moves through a copy of the state space (thereby producing a path π), and the \exists -player reacts with moves in a separate copy (thereby producing a path π'). The \exists -player wins if π' combined with π satisfies φ , in which case the property holds.⁷ In HyperATL^{*}, we can express this game-based verification approach as a logical statement: Formula $[\forall\pi. \exists\pi'] \varphi$ requires a strategy that constructs π' when played in parallel with a game that constructs π . A system thus satisfies $[\forall\pi. \exists\pi'] \varphi$ exactly if the property can be verified in the game-based approach from [CFST19].

Phrased differently, while [CFST19] derives a verification *method*, HyperATL^{*} can express both the original HyperLTL property and its game-based verification method as *formulas*; the correctness of the algorithm from [CFST19] becomes a logical implication in HyperATL^{*}. Lemma 4.3 states a more general implication (by considering a property of the form $\forall\pi. \langle\langle A \rangle\rangle\pi'. \varphi$ instead of $\forall\pi. \exists\pi'. \varphi$).

Lemma 4.3. *Let \mathcal{G} be any game structure and φ be any HyperATL^{*} formula. If $\mathcal{G} \models [\forall\pi. \langle\langle A \rangle\rangle\pi'] \varphi$ then $\mathcal{G} \models \forall\pi. \langle\langle A \rangle\rangle\pi'. \varphi$.*

Proof. Assume that $\mathcal{G} \models [\forall\pi. \langle\langle A \rangle\rangle\pi'] \varphi$. Let $F_A = \{f_\xi : (S \times S)^+ \rightarrow \mathcal{M} \mid \xi \in A\}$ be the set of strategies for the agents in A that is winning, *i.e.*, for every $(p, p') \in \text{out}(\mathcal{G}, (s_0, s_0), \emptyset, F_A)$ we have $[\pi \mapsto p, \pi' \mapsto p'] \models \varphi$. We show that $\mathcal{G} \models \forall\pi. \langle\langle A \rangle\rangle\pi'. \varphi$. Let $\Pi = [\pi \mapsto p]$ be any path assignment for π (which is universally quantified). We construct a winning strategy $f'_\xi : S^+ \rightarrow \mathcal{M}$ for each $\xi \in A$. For $u \in S^+$ we define

$$f'_\xi(u) := f_\xi(p[0, |u| - 1] \otimes u).$$

Strategy f'_ξ disregards most of the already fixed path p and queries f_ξ on prefixes of p . Let $F'_A = \{f'_\xi \mid \xi \in A\}$. It is easy to see that for each $p' \in \text{out}(\mathcal{G}, s_0, F'_A)$ we have $p \otimes p' \in \text{out}(\mathcal{G}, (s_0, s_0), \emptyset, F_A)$ and so $\mathcal{G} \models \forall\pi. \langle\langle A \rangle\rangle\pi'. \varphi$ as required. \square

Using Lemma 4.3 (which generalizes easily to formulas of the form $\forall\pi_1 \dots \forall\pi_k. \langle\langle A \rangle\rangle\pi'. \varphi$), we can, for example, strengthen *GNI* by surrounding the quantifier prefix with $[\]$ brackets. We can manually increase the lookahead to enable the strategy that is constructing path π' to peek at future events on π_1, \dots, π_k by shifting the system.

Definition 4.4. For a game structure $\mathcal{G} = (S, s_{\text{init}}, \Xi, \mathcal{M}, \delta, L)$ and $n \geq 1$ we define

$$\text{shift}(\mathcal{G}, n) := (S \cup \{s_0, \dots, s_{n-1}\}, s_0, \Xi, \mathcal{M}, \delta', L')$$

where s_0, \dots, s_{n-1} are fresh states not already in S . The transition function δ' is defined by $\delta'(s, \sigma) := \delta(s, \sigma)$ for $s \in S$, $\delta'(s_i, \sigma) := s_{i+1}$ if $i < n - 1$ and $\delta'(s_{n-1}, \sigma) := s_{\text{init}}$. The labeling function L' is defined by $L'(s) = L(s)$ if $s \in S$ and $L'(s_i) = \emptyset$. We define $\text{shift}(\mathcal{G}, 0) := \mathcal{G}$. \triangleleft

⁷The reverse implication does, in general, not hold as the \exists -player might require knowledge about the future behavior of the \forall -player. The game-based verification method can be made complete by adding *prophecies*, *i.e.*, hints for the \exists -player that provide limited information about the future behavior of the \forall -player [BF22a].

System $shift(\mathcal{G}, n)$ shifts the behavior of \mathcal{G} by adding n initial steps before continuing as in \mathcal{G} . We express a shifted approximation of GNI as follows.

$$\left[\forall_{\mathcal{G}} \pi. \forall_{\mathcal{G}} \pi'. \exists_{shift(\mathcal{G}, n)} \pi'' \right] \square \left(\bigwedge_{a \in H} a_{\pi} \leftrightarrow \bigcirc^n a_{\pi''} \right) \wedge \square \left(\bigwedge_{a \in L \cup O} a_{\pi'} \leftrightarrow \bigcirc^n a_{\pi''} \right) \quad (approxGNI_n)$$

We shift the behavior of the copy on which π'' is constructed by n positions which we correct using n \bigcirc s in the body of the formula. It is easy to see that if $\mathcal{G} \models approxGNI_n$ for some n , then $\mathcal{G} \models GNI$.⁸

4.3. Simulation-based Non-Interference. The previously discussed information-flow policies are path-based. By contrast, *simulation*-based definitions of non-interference require a lock-step security simulation that disproves leakage (see, *e.g.*, [SS00, Sab03, MS10]). Let $\mathcal{G} = (S, s_0, \{\xi_N, \xi_L, \xi_H\}, \mathcal{M}, \delta, L)$ be a progCGS. For states $s, s' \in S$ and evaluations $i_L \in 2^L$ and $i_H \in 2^H$, we write $s \Rightarrow_{i_H}^{i_L} s'$ if $L(s') \cap L = i_L$ and $L(s') \cap H = i_H$ and s' is a possible successor of s in \mathcal{G} (*i.e.*, there exists a move vector σ such that $\delta(s, \sigma) = s'$).

Definition 4.5. A *security simulation* is a relation $R \subseteq S \times S$ such that whenever $(s, t) \in R$, we have 1) s and t agree on the output propositions, *i.e.*, $L(s) \cap O = L(t) \cap O$, and 2) for any $i_L \in 2^L$ and $i_H, i'_H \in 2^H$ if $s \Rightarrow_{i_H}^{i_L} s'$ then there exists a t' with $t \Rightarrow_{i'_H}^{i_L} t'$ and $(s', t') \in R$. \triangleleft

Note that this is not equivalent to the fact that R is a simulation in the standard sense [Mil80] as the second condition is asymmetric in the high-security inputs. We call \mathcal{G} *simulation secure* if there exists a security simulation R with $(s_0, s_0) \in R$ [Sab03, SS00]. It is easy to see that every input-total system that is *simulation secure* satisfies GNI . The converse does, in general, not hold. In HyperATL*, we can express simulation security.

$$\left[\forall_{\mathcal{G}} \pi. \langle \xi_N \rangle_{shift(\mathcal{G}, 1)} \pi' \right] \square \left(\bigwedge_{a \in L} a_{\pi} \leftrightarrow \bigcirc a_{\pi'} \right) \rightarrow \square \left(\bigwedge_{a \in O} a_{\pi} \leftrightarrow \bigcirc a_{\pi'} \right) \quad (simSec)$$

Here we shift the path π' by one position ($shift(\mathcal{G}, 1)$), which we correct using the \bigcirc . The shifting allows the strategy for ξ_N in the second copy to base its decision on an already fixed step in the first copy, *i.e.*, it corresponds to a strategy with a fixed lookahead of 1 step.

Lemma 4.6. A progCGS \mathcal{G} is simulation secure if and only if $\mathcal{G} \models simSec$.

Proof. We only sketch the high-level idea as the proof is similar to the well-known characterization of simulations and bisimulations as two-player games [Sti95]. For the first direction, we assume that \mathcal{G} is *simulation secure* and let R be a security simulation witnessing this. The idea of the strategy for ξ_N is to choose successors such that the parallel game between both copies is always in R -related states (after shifting). This is possible as long as the premise of *simSec* is not violated. By the definition of security simulations, this already implies $\square(\bigwedge_{a \in O} a_{\pi} \leftrightarrow \bigcirc a_{\pi'})$. Note that the shifting is key, so the strategy for ξ_N fixes a move *after* the universally quantified path chooses a successor (as in the definition of security simulation). For the second direction, assume ξ_N has a winning strategy. We construct the relation R by defining $(s, t) \in R$ whenever the pair (s, t) occurs in the (shifted) parallel composition in any play for which the premise of *simSec* holds. \square

⁸This implication is of practical relevance as *approxGNI_n* sits at a fragment of HyperATL* checkable in polynomial time (in the size of the system), whereas the $\forall^* \exists^*$ -fragment of HyperLTL (and thus HyperATL*) is already PSPACE-hard [Rab16].

4.4. Non-Deducibility of Strategies. As the last example in this section, we consider *non-deducibility of strategies (NDS)* [WJ90]. *NDS* requires that every possible output is compatible with every possible input-*strategy* (whereas *GNI* requires it to be compatible with every possible input-*sequence*). The subtle difference between sequences and strategies is important when a high-security input player can observe the internal state of a system. As a motivating example, consider the following (first introduced in [WJ90]):

Example 4.7. Suppose we have a system that reads a binary input h from a high-security source and outputs o . The system maintains a bit b of information in its state, initially chosen non-deterministically. In each step, the system reads the input h , outputs $h \oplus b$ (where \oplus is the xor-operation), non-deterministically picks a new value for b and then repeats. As \oplus encodes an one-time pad, it is not hard to see that this system satisfies *GNI*: Given any input, any output is possible by resolving the non-deterministic choice of b appropriately.

If the input player is, however, able to observe the system (in the context of [WJ90] the system shares the internal bit on a private channel), she can communicate an arbitrary sequence of bits to the low-security environment. Whenever she wants to send bit c , she inputs $h = c \oplus b$ where b is the value of the internal bit (note that $(c \oplus b) \oplus b = c$). \triangleleft

Instead of requiring that every output sequence is compatible with all possible high-security input sequences, we require it to be compatible with all possible high-security input strategies. Phrased differently, there should *not* be an output sequence such that a strategy for the input-player can *avoid* this output.

$$\neg \exists \pi. \langle \langle \xi_H \rangle \rangle \pi'. \Box (\bigwedge_{a \in L} a_\pi \leftrightarrow a_{\pi'}) \rightarrow \Diamond (\bigvee_{a \in O} a_\pi \not\leftrightarrow a_{\pi'}) \quad (NDS)$$

This formula states that there does not exist a path π such that ξ_H has a strategy to avoid the output of π (provided with the same low-security inputs). The system sketched in Example 4.7 does not satisfy *NDS* (there, *e.g.*, exists a strategy for ξ_H that ensures that the output o is always set to true). Note that, due to determinacy of parity games, *NDS* is equivalent to *stratNI* on turn-based game structures.

5. STRATEGIC AND ASYNCHRONOUS HYPERPROPERTIES

Most existing hyperlogics traverse the paths of a system synchronously. However, especially when reasoning about software systems, one requires an asynchronous traversal to account, for example, for the unknown execution speed or to abstract away from intermediate (non-observable) program steps. In this section, we outline how strategic hyperproperties are useful to express such asynchronous hyperproperties.

The idea is to express asynchronous hyperproperties by viewing the stuttering of a system (*i.e.*, whether a system progresses or remains in its current state) as being resolved by a dedicated player (which we call scheduling player). Quantification over strategies of the scheduling player then naturally corresponds to asynchronous reasoning. That is, instead of reasoning about the (asynchronous) scheduling of a system directly, we reason about strategies for the scheduling agent. This style of asynchronous reasoning can express many properties while remaining fully decidable (as model checking of **HyperATL*** is decidable, see Section 6) and yields formulas that are automatically checkable (see Section 8).

5.1. Scheduling Player and Stuttering Transformation. We call a player *sched* an asynchronous scheduler if it can decide whether the system progresses (as decided by the other agents) or stutters. Note that this differs from the asynchronous turn-based games as defined in [AHK02]. In our setting, the scheduler does not control which of the player controls the next move but rather decides if the system as a whole progresses or stutters. In cases where the system does not already include an asynchronous scheduler, we can include a scheduler via a simple system transformation.

Definition 5.1. Given a game structure $\mathcal{G} = (Q, q_0, \Xi, \mathcal{M}, \delta, d, L)$ over AP and a fresh agent *sched* not already included in Ξ , define the stutter version of \mathcal{G} , denoted \mathcal{G}_{stut} , as the game structure over $AP \cup \{stut\}$ by $\mathcal{G}_{stut} := (Q \times \{0, 1\}, (q_0, 0), \Xi \cup \{sched\}, \mathcal{M} \times \{\rightarrow, \Downarrow\}, \delta', d', L')$ where

$$\delta'((s, -), \sigma) := \begin{cases} (\delta(s, proj_1 \circ \sigma|_{\Xi}), 0) & \text{if } (proj_2 \circ \sigma)(sched) = \rightarrow \\ (s, 1) & \text{if } (proj_2 \circ \sigma)(sched) = \Downarrow \end{cases}$$

$L'((s, 0)) := L(s)$ and $L'(s, 1) := L(s) \cup \{stut\}$. Finally $d'(\xi) := d(\xi)$ for $\xi \in \Xi$ and $d'(sched) := m + 1$ where m is the maximal element in the codomain of d . \triangleleft

Here, $proj_i$ is the projection of the i th element in a tuple, \circ denotes function composition, and $-$ represents an arbitrary value in that position. In \mathcal{G}_{stut} , the agents of the original game structure \mathcal{G} , fix moves in \mathcal{M} and thereby determine the next state of the system. In addition, the $\{\rightarrow, \Downarrow\}$ -decision of scheduling player determines if the move is actually executed (\rightarrow) or if the system remains in its current state (\Downarrow). As *sched* sits in the last stage of the MSCGS, the scheduling decision is based on the already fixed moves of the agents in Ξ . The extended state-space $Q \times \{0, 1\}$ is used to keep track of the stuttering, which becomes visible via the new atomic proposition *stut*.

5.2. Observational Determinism. As a warm-up, we again consider the property of observational-determinism which states that the output along all paths is identical, *i.e.*, $\forall \pi. \forall \pi'. \Box (\bigwedge_{a \in O} a_\pi \leftrightarrow a_{\pi'})$. We already argued that the example program in the introduction (in Figure 2) does not satisfy this property (if interpreted as a transition system in the natural way), as the output changes at different time points. To express an asynchronous version of OD, we reason about (a strategy for) the scheduling player on the transformed system.

$$[\llbracket sched \rrbracket \pi. \llbracket sched \rrbracket \pi'] \text{fair}_\pi \wedge \text{fair}_{\pi'} \wedge \Box \left(\bigwedge_{a \in O} a_\pi \leftrightarrow a_{\pi'} \right) \quad (OD_{asynch})$$

where $\text{fair}_\pi := \Box \Diamond \neg \text{stut}_\pi$, asserts that the system may not be stuttered forever. Note that we encapsulated the quantifiers by $[\cdot]$, thus resolving the games in parallel. Note that *sched* only controls the stuttering and not the path of the underlying system. The example from Figure 2, after the stuttering transformation, satisfies this formula, as the output can be aligned by the scheduling player.

5.3. One-Sided Stuttering. By resolving the stuttered paths incrementally (*i.e.*, omitting the $[\cdot]$ -brackets), we can also express *one-sided stuttering*, *i.e.*, allow only the second copy to be stuttered. As an example, assume P^h is a program written in a higher-level programming language and P^l the compiled program into a low-level language (*e.g.*, assembly code). Let \mathcal{T}^h and \mathcal{T}^l be transition systems of both programs, and consider the property that the low-level program exhibits the same output as the original program. As the compiler breaks

each program statement into multiple low-level instructions, the outputs will not match in a synchronous manner. Instead, the system \mathcal{T}^h may need to stutter for the low-level program to “catch up”. Using Definition 5.1 we can express this as follows.

$$\forall_{\mathcal{T}^l} \pi. \llbracket sched \rrbracket_{\mathcal{T}^h} \pi'. fair_{\pi'} \wedge \square \left(\bigwedge_{a \in O} a_{\pi} \leftrightarrow a_{\pi'} \right)$$

i.e., for every execution π of low-level program we can stutter the high-level program such that the observations align.

5.4. Asynchronous HyperLTL. We compare the strategic approach to asynchronicity of HyperATL* (in Section 5.2) with *asynchronous HyperLTL* (AHLTL for short) [BCB⁺21], a recent extension of HyperLTL specifically designed to express asynchronous properties. AHLTL is centered around the stuttering of a path. A path p' is a stuttering of p , written $p \preceq p'$, if it is obtained by stuttering each step in p finitely often. Formulas in AHLTL quantify universally or existentially over stuttering of paths. For example, the AHLTL formula $\forall \pi_1 \dots \forall \pi_n. \mathbf{E}. \varphi$ holds on a transition system \mathcal{T} (written $\mathcal{T} \models_{\text{AHLTL}} \forall \pi_1 \dots \forall \pi_n. \mathbf{E}. \varphi$) if for all path p_1, \dots, p_n in the \mathcal{T} , there exists stutterings p'_1, \dots, p'_n (*i.e.*, $p_i \preceq p'_i$ for all i) that (when bound to π_1, \dots, π_n) satisfy φ . Different from the asynchronous treatment in HyperATL*, the stuttering in AHLTL is thus quantified *after* all paths are fixed.

Finite-state model checking of AHLTL is undecidable, already for formulas of the form $\forall \pi_1. \forall \pi_2. \mathbf{E}. \varphi$ [BCB⁺21]. The largest known fragment of AHLTL with decidable model checking problem are formulas of the form $\forall \pi_1 \dots \forall \pi_n. \mathbf{E}. \varphi$ where φ is an *admissible formula*. An admissible formula has the form

$$\varphi = \varphi_{state} \wedge \left(\bigwedge_{i=1}^n \varphi_{stut}^i \right) \wedge \varphi_{phase}$$

where φ_{state} is a state-formula, *i.e.*, uses no temporal operators, each φ_{stut}^i is a stutter invariant formula that only refers to a single path variable, and φ_{phase} is a phase formula which is a conjunction of formulas of the form $\square \bigwedge_{a \in P} (a_{\pi_i} \leftrightarrow a_{\pi_j})$. The phase formula requires the two paths π_i and π_j to traverse the same sequence (phases) of “colors” (as defined by P). See [BCB⁺21, §4.1] for a more detailed discussion.

By replacing the stuttering quantifier \mathbf{E} in AHLTL with strategy quantification in HyperATL* we obtain a sound approximation for formulas of the form $\forall \pi_1 \dots \forall \pi_n. \mathbf{E}. \varphi$.

Theorem 5.2. *Assume a transition system \mathcal{T} and AHLTL formula $\forall \pi_1 \dots \forall \pi_n. \mathbf{E}. \varphi$. If*

$$\mathcal{T}_{stut} \models [\llbracket sched \rrbracket \pi_1 \dots \llbracket sched \rrbracket \pi_n] \varphi \wedge \bigwedge_{i=1}^n fair_{\pi_i} \quad (5.1)$$

then

$$\mathcal{T} \models_{\text{AHLTL}} \forall \pi_1 \dots \forall \pi_n. \mathbf{E}. \varphi. \quad (5.2)$$

If $n = 2$ and φ is an admissible formula, 5.1 and 5.2 are equivalent.

Proof. Let $\mathcal{T} = (S, s_0, \delta, L)$. We first show that 5.1 implies 5.2. Assume 5.1 and let $f_{sched}^i : (S^n)^+ \rightarrow \mathcal{M} \times \{\rightarrow, \downarrow\}$ for $1 \leq i \leq n$ be a winning strategy for the scheduler. To show 5.2, let p_1, \dots, p_n be any paths in \mathcal{T} . For each $1 \leq i \leq n$ we define a stuttered version p'_i such that $p_i \preceq p'_i$. As intermediate steps we define $p_{1,m}, \dots, p_{n,m} \in S^m$ and $c_{1,m}, \dots, c_{n,m} \in \mathbb{N}$ for each $m \in \mathbb{N}_{\geq 1}$ by recursion on m . For $m = 1$, we set $p_{i,1} = p_i(0)$, *i.e.*, a path of length 1 and $c_{i,1} = 1$ for each $1 \leq i \leq n$. For $m > 1$, define $b_{i,m} := proj_2 \circ f_{sched}^i(\otimes(p_{1,m-1}, \dots, p_{n,m-1}))$ for

each $1 \leq i \leq n$. If $b_{i,m} = \Rightarrow$, we define $c_{i,m} := c_{i,m-1} + 1$ and if $b_{i,m} = \Downarrow$ we define $c_{i,m} := c_{i,m-1}$. We then set $p_{i,m} := p_{i,m} \cdot p_i(c_{i,m})$ (where \cdot denotes sequence concatenation). Define $p'_i \in S^\omega$ as the limit of $\{p_{i,m}\}_{m \in \mathbb{N}_{\geq 1}}$ (which exists as $p_{i,m}$ is a prefix of $p_{i,m+1}$ for every m). It is easy to see that $p_i \sqsubseteq p'_i$ (the fairness assumption in 5.1 ensures that a path is not stuttered forever). Moreover $[\pi_1 \mapsto p'_1, \dots, \pi_n \mapsto p'_n] \models \varphi$ holds as $\{f_{sched}^i\}_{i=1}^n$ is winning, and so $[\pi_1 \mapsto p_1, \dots, \pi_n \mapsto p_n] \models_{\text{AHLTL}} \mathbf{E}.\varphi$ as required.

For the second direction, assume that 5.2 holds and that φ is admissible. Let $\varphi = \varphi_{state} \wedge (\bigwedge_{i=1}^l \varphi_{stut}^i) \wedge \varphi_{phase}$. State formula φ_{state} only refers to the initial states (as it is free of temporal operators) and φ_{stut}^i is, by assumption, stutter invariant, so any fair scheduling chosen by *sched* satisfies both φ_{state} and $\bigwedge_{i=1}^l \varphi_{stut}^i$. Let $\varphi_{phase} = \square \bigwedge_{a \in P} (a_{\pi_1} \leftrightarrow a_{\pi_2})$ be the phase formula. We say two states $s, s' \in S$ are *about to change phase* if for some $a \in P$ we have $a \in L(s) \not\leftrightarrow a \in L(s')$. In this case, we write $change(s, s')$. The strategy for the scheduler has access to both the current state s_1, s_2 of both copies and the next state s'_1, s'_2 in both copies (as the scheduler sits at the last stage of \mathcal{T}_{stut}). The joint strategy for the scheduler in both copies behaves as follows: If $change(s_1, s'_1) \leftrightarrow change(s_2, s'_2)$ holds, *i.e.*, either both or none of the copies are about to change phase, it schedules *both* copies. Otherwise, it only schedules the copy that is *not* about to change phase. This ensures that phase changes occur synchronized in both copies. As we assumed 5.2, there is a stuttering for all paths in the system such that φ_{phase} holds, *i.e.*, all paths traverse the same phases, albeit at possibly different speeds. It is easy to see that the strategy defined above creates an alignment into identical phases for any two paths of the system. Consequently, any play compatible with this strategy satisfies φ_{phase} (and therefore also φ), and so 5.1 holds as required. \square

Theorem 5.2 gives a sound approximation of the (undecidable) AHLTL model checking that is exact for admissible formulas. As HyperATL* model checking is decidable (see Section 6) and the stuttering construction \mathcal{T}_{stut} is effectively computable, we derive an alternative proof of the decidability result from [BCB⁺21]. In summary, HyperATL* subsumes the largest (known) decidable fragment of AHLTL, while enjoying decidable model checking for the full logic (see the following Section 6). Moreover, the HyperATL* formula 5.1 constructed in Theorem 5.2 falls in the fragment supported by our model checker (see Section 8).

6. AUTOMATA-BASED MODEL CHECKING

In this section, we present an automata-based model checking algorithm for HyperATL*, *i.e.*, given a formula $\dot{\varphi}$ (we use the dot to refer to the original formula and use φ, ψ to refer to sub-formulas of $\dot{\varphi}$) and a game structure $\mathcal{G} = (S, s_0, \Xi, \mathcal{M}, \delta, d, L)$ we decide if $\mathcal{G} \models \dot{\varphi}$. Before discussing our verification approach, let us briefly recall ATL* model checking [AHK02] and why the approach is not applicable to HyperATL*. In ATL*, checking if $\langle\langle A \rangle\rangle \varphi$ holds in some state s can be reduced to the non-emptiness check of the intersection of two tree automata. One accepting all possible trees that can be achieved via a strategy for players in A starting in s , and one accepting all trees whose paths satisfy the path formula φ [AHK02]. In our hyperlogic, this is not possible. When checking $\langle\langle A \rangle\rangle \pi.\varphi$, we cannot construct an automaton accepting all trees that satisfy φ , as the satisfaction of φ depends on the paths assigned to the outer path-quantifiers (which are not yet fixed).

Instead, we construct an automaton that accepts all *path assignments* for the outer quantifiers for which there exists a winning strategy for the agents in A (similar to the model checking approach for HyperCTL* [FRS15]). Different from the approach for HyperCTL*, we

cannot resolve path quantification via an existential or universal product construction and instead encode the strategic behavior of \mathcal{G} within the transition function of an *alternating* automaton.

In the following, we 1) define a notion of equivalence between formulas and automata and discuss the overall model checking algorithm (in Section 6.1), 2) give an inductive construction of an equivalent automaton (in Section 6.2), 3) prove the construction correct (in Section 6.3), and 4) discuss the complexity of our algorithm (in Section 6.4).

6.1. \mathcal{G} -Equivalence and Model Checking Algorithm. Recall that for paths $p_1, \dots, p_n \in S^\omega$, $\otimes(p_1, \dots, p_n) \in (S^n)^\omega$ denotes the pointwise product (also called the zipping). Assume that some HyperATL* formula ψ contains free path variables π_1, \dots, π_n (in our algorithm ψ is a sub-formula of $\dot{\varphi}$ that occurs under path quantifiers that bind π_1, \dots, π_n). We say that an automaton \mathcal{A} over alphabet $\Sigma_\psi := S^n$ is *\mathcal{G} -equivalent* to ψ , if for any paths p_1, \dots, p_n it holds that

$$[\pi_i \mapsto p_i]_{i=1}^n \models_{\mathcal{G}} \psi \quad \text{iff} \quad \otimes(p_1, \dots, p_n) \in \mathcal{L}(\mathcal{A}).$$

That is, \mathcal{A} accepts a zipping of paths exactly if the path assignment constructed from those paths satisfies the formula; \mathcal{A} summarizes all path assignments for the free variables that satisfy a formula.

Now let $\dot{\varphi}$ be the formula to be checked. Our model checking algorithm progresses in an (inductive) bottom-up manner and constructs an automaton \mathcal{A}_ψ that is \mathcal{G} -equivalent for each subformula ψ of $\dot{\varphi}$ (we give the construction in the next section). Consequently, we obtain an automaton $\mathcal{A}_{\dot{\varphi}}$ over alphabet $\Sigma_{\dot{\varphi}} = S^0$ that is \mathcal{G} -equivalent to $\dot{\varphi}$. By definition of \mathcal{G} -equivalence, $\mathcal{A}_{\dot{\varphi}}$ is non-empty iff $\emptyset \models_{\mathcal{G}} \dot{\varphi}$ iff $\mathcal{G} \models \dot{\varphi}$. As emptiness of alternating parity automata is decidable [MH84, BKR10] we can decide whether $\mathcal{G} \models \dot{\varphi}$.

6.2. Construction of \mathcal{G} -Equivalent Automata. In the following, we give a construction of a \mathcal{G} -equivalent automaton for each syntactic construct of HyperATL*. The most interesting case is the construction for a formula $\varphi = \langle\langle A \rangle\rangle \pi. \psi$ where we construct an automaton \mathcal{A}_φ over $\Sigma_\varphi = S^n$ from an automaton \mathcal{A}_ψ over $\Sigma_\psi = S^{n+1}$ by a suitable product construction with \mathcal{G} that takes the strategic behavior in the game structure into account. We split the construction into the cases of logical and temporal operators (in Figure 3), simple quantification (in Figure 4), and complex quantification (in Figure 5).

Boolean And Temporal Operators. For the boolean combinators and temporal operators, our construction follows the standard translation from LTL to alternating automata (see, e.g., [MSS88] or [FRS15] for details). We give the construction in Figure 3.

Simple Quantification. We now consider the case where $\varphi = \langle\langle A \rangle\rangle \pi. \psi$ and focus on the case where the quantifier is *simple*. Assume that $A = \Xi$, i.e., $\varphi = \exists \pi. \psi$. The construction of \mathcal{A}_φ is similar to the one in [FRS15, BF23a] by building a product of \mathcal{A}_ψ and \mathcal{G} . We give the construction in Figure 4. Here \mathcal{A}_ψ^{ndet} is a *non-deterministic* automaton equivalent to \mathcal{A}_ψ , which we can obtain (with an exponential blowup) via Theorem 2.3. The automaton \mathcal{A}_φ guesses a path in \mathcal{G} and tracks the acceptance of \mathcal{A}_ψ on this path combined with the input word over S^n , i.e., every accepting run of \mathcal{A}_φ on $\otimes(p_1, \dots, p_n)$ guesses a path p in \mathcal{G} such that \mathcal{A}_ψ accepts $\otimes(p_1, \dots, p_n, p)$. Note that in this case \mathcal{A}_φ is again a non-deterministic automaton. The case where $A = \emptyset$ (i.e., $\varphi = \forall \pi. \psi$) can be handled using complementation:

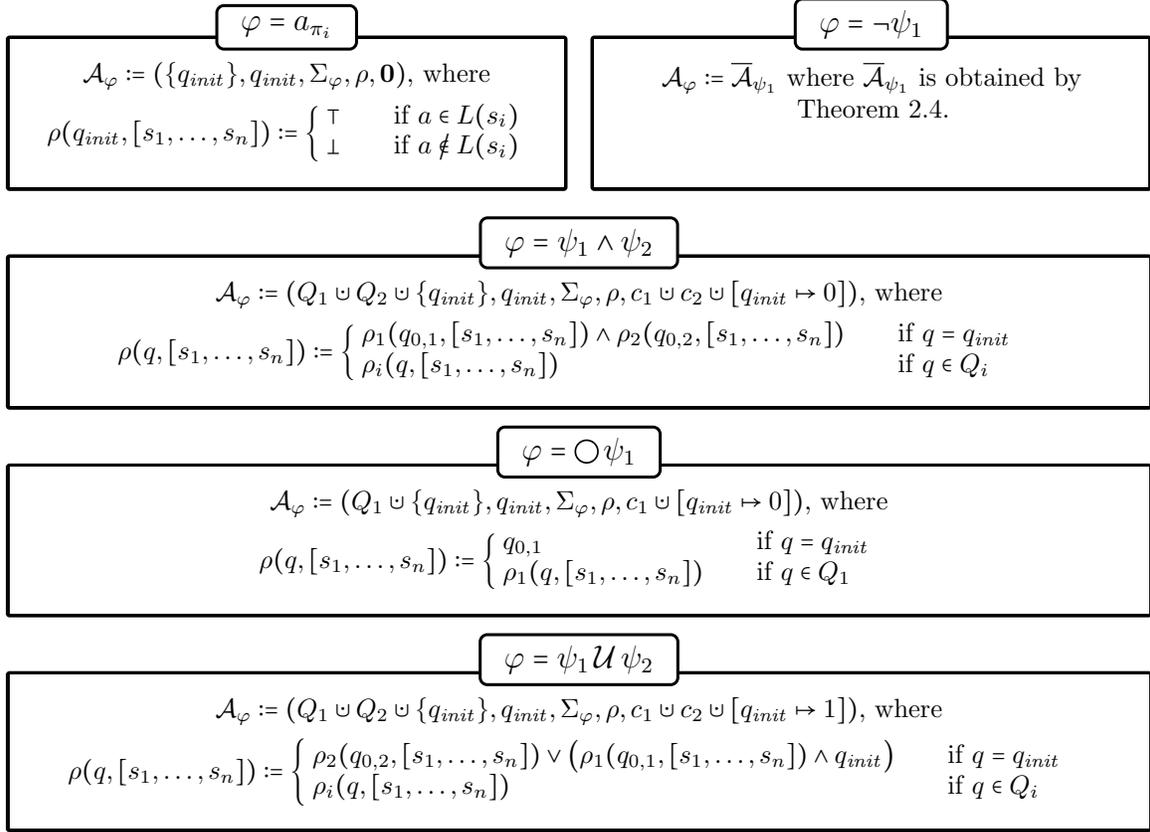


FIGURE 3. Automaton construction for boolean and temporal operators. Here, $\mathcal{A}_{\psi_i} = (Q_i, q_{0,i}, \Sigma_{\psi_i}, \rho_i, c_i)$ for $i \in \{1, 2\}$ are inductively constructed alternating automata for sub-formulas ψ_1 and ψ_2 . We assume that Q_1 and Q_2 are disjoint sets of states and q_{init} is a fresh state. For two colorings $c_1 : Q_1 \rightarrow \mathbb{N}$ and $c_2 : Q_2 \rightarrow \mathbb{N}$, $c_1 \cup c_2 : Q_1 \cup Q_2 \rightarrow \mathbb{N}$ denotes the combined coloring. We write $[q \mapsto n]$ for the function $\{q\} \rightarrow \mathbb{N}$ that maps q to n .

As $\forall\pi. \varphi \equiv \neg\exists\pi. \neg\varphi$ we can combine the construction for existential quantification (in Figure 4) with the construction for negation in Figure 3. Importantly, in cases where $A = \emptyset$, the automaton \mathcal{A}_φ is universal.

Strategic Quantification. Lastly, we consider the case of (proper) complex quantification, *i.e.*, the case where $\varphi = \langle\langle A \rangle\rangle\pi. \psi$ and $A \neq \emptyset$ and $A \neq \exists$.⁹ In our construction, \mathcal{A}_φ encodes the strategic behavior of the agents in A . We achieve this by encoding the strategic play of the game structure within the transition function of \mathcal{A}_φ . We give the construction in Figure 5. Here \mathcal{A}_ψ^{det} is a *deterministic* automaton equivalent to \mathcal{A}_ψ which we obtain (with a double exponential blowup) via Theorem 2.3. The transition function encodes the strategic behavior by disjunctively choosing moves for players in A , followed by a conjunctive

⁹Note that the construction in Figure 5 subsumes the construction in Figure 4. We give an explicit construction for the case of simple quantification (in Figure 4) as the resulting automaton is exponentially smaller, giving tight complexity results (see Section 6.4).

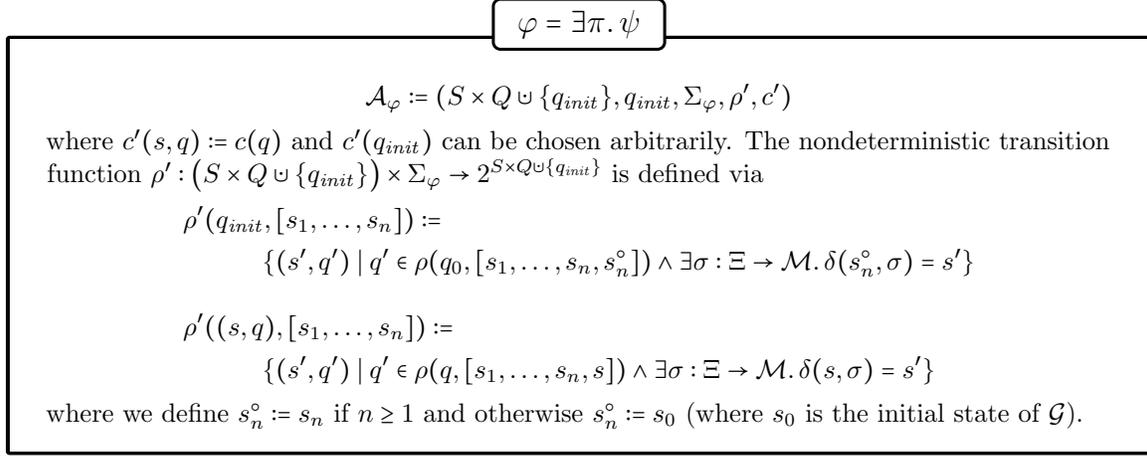


FIGURE 4. Construction of a \mathcal{G} -equivalent automaton for $\varphi = \exists \pi. \psi$. Here, $\mathcal{A}_\psi^{ndet} = (Q, q_0, \Sigma_\psi, \rho, c)$ with $\rho : Q \times \Sigma_\psi \rightarrow 2^Q$ is a non-deterministic automaton that is equivalent to the inductively constructed automaton \mathcal{A}_ψ for ψ .

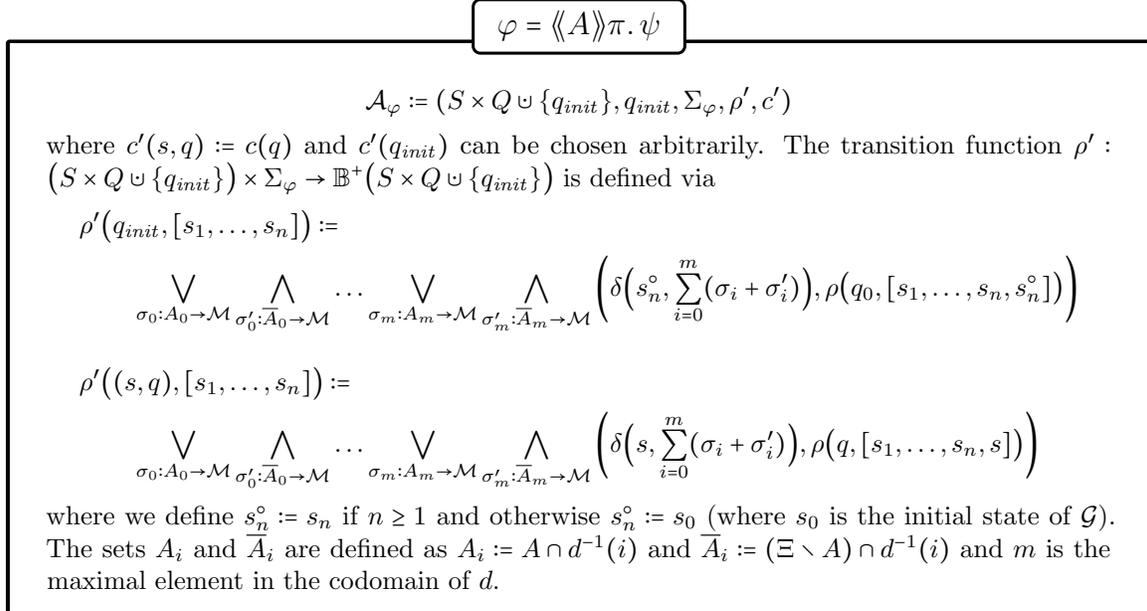


FIGURE 5. Construction of a \mathcal{G} -equivalent automaton for $\varphi = \langle\langle A \rangle\rangle \pi. \psi$. Here, $\mathcal{A}_\psi^{det} = (Q, q_0, \Sigma_\psi, \rho, c)$ with $\rho : Q \times \Sigma_\psi \rightarrow Q$ is a deterministic automaton that is equivalent to the inductive constructed alternating automaton \mathcal{A}_ψ for ψ .

treatment of all adversarial players. The stages of \mathcal{G} naturally correspond to the order of the move selection (as captured in the sets A_i and \bar{A}_i), giving an alternating sequence of disjunctions and conjunctions. In the case where the MSCGS \mathcal{G} is a CGS, *i.e.*, $d = \mathbf{0}$, the transition function has the form of a (positive) DNF (a boolean formula of the form $\bigvee \bigwedge$),

where the moves of agents in A are considered disjunctively and the moves by all other agents conjunctively. Our construction can be extended to handle formulas of the form $[\langle\langle A_1 \rangle\rangle\pi_1 \dots \langle\langle A_k \rangle\rangle\pi_k] \psi$ by joining the stage across k copies of the game structure. For the dual strategic quantifier $\llbracket A \rrbracket \pi. \psi$ we can again make use of the fact that $\llbracket A \rrbracket \pi. \psi \equiv \neg \langle\langle A \rangle\rangle. \neg \psi$ and combine the construction in Figure 5 with that for negation in Figure 3.

6.3. Correctness. The correctness of our construction in Section 6.2 is stated by the following proposition.

Proposition 6.1. *For any HyperATL* formula φ , \mathcal{A}_φ is \mathcal{G} -equivalent to φ .*

The proof of Proposition 6.1 goes by induction on φ following the construction of \mathcal{A}_φ . For the logical and temporal connectives (in Figure 3) and pure existential quantification (in Figure 4), the statement is obvious (see, *e.g.*, [MSS88] for the logical and temporal connectives and [FRS15] for the case of simple quantification). A proof for the case where $\varphi = \langle\langle A \rangle\rangle \pi. \psi$ (in Figure 5) can be found in Appendix A.

6.4. Complexity Upper Bounds. The complexity of our model checking algorithm hinges on the size of the automaton \mathcal{A}_φ . The constructions in Figure 3 only increase the size of the automaton by a polynomial amount, but the constructions for path quantification in Figures 4 and 5 increase the number of states exponentially. We observe a difference in blowup between simple quantification and complex quantification. The former requires (in general) a conversion of the alternating automaton \mathcal{A}_ψ to a non-deterministic automaton (\mathcal{A}_ψ^{ndet} in Figure 4) causing an exponential blowup, whereas the latter requires a full determinization (\mathcal{A}_ψ^{det} in Figure 5) causing a *double* exponential blowup.

To capture the size of the automaton and the resulting complexity of our algorithm, we define $Tower_c(k, n)$ as a tower of k exponents (with base c), *i.e.*,

$$\begin{aligned} Tower_c(0, n) &:= n \\ Tower_c(k+1, n) &:= c^{Tower_c(k, n)} \end{aligned}$$

For $k \geq 1$, we define k -EXPSpace as the class of languages recognized by a deterministic (or, due to Savitch's theorem [Sav70], equivalently, non-deterministic) Turing machine (TM) with space $Tower_c(k, n)$ for some fixed $c \in \mathbb{N}$. Analogously, we define k -EXPTIME as the class of languages recognizable by a *deterministic* TM in time $Tower_c(k, n)$ for some fixed c . We define 0 -EXPTIME := PTIME and 0 -EXPSpace := PSPACE. For $k < 0$, we define k -EXPSpace := NLOGSPACE.

Complexity Based on Prefix-Cost. We characterize the cost of our algorithm for HyperATL* model checking. We distinguish between the complexity in the size of the specification (the length of the formula, *i.e.*, the number of nodes in the AST) and the size of the system (the number of states). Our complexity analysis is parametric in the structure of the quantifier prefix of a formula.¹⁰ We focus our discussion on HyperATL* formulas

¹⁰If we consider arbitrary HyperATL* formulas, model checking is non-elementary in both the size of the specification and the size of the system (as HyperATL* subsumes HyperLTL; see [Rab16] for details). If we instead consider formulas with a fixed quantifier structure, we can derive elementary complexity bounds (in terms of specification size and system size) for all formulas with the fixed quantifier structure.

$$\begin{aligned}
d_{spec}(\exists\pi.\psi) &:= 1 \\
d_{spec}(\forall\pi.\psi) &:= 1 \\
d_{spec}(\langle\langle A \rangle\rangle\pi.\psi) &:= 2 \\
d_{spec}(\llbracket A \rrbracket\pi.\psi) &:= 2 \\
d_{spec}(\mathbb{Q}_1\pi.\mathbb{Q}_2\pi'.\varphi) &:= d_{spec}(\mathbb{Q}_2\pi'.\varphi) + q(\mathbb{Q}_1, \mathbb{Q}_2)
\end{aligned}$$

(A) Prefix-cost w.r.t. the size of the *specification*. Here, ψ is the quantifier-free body of the formula. The definition of the cost function q is given in Figure 6c.

$$\begin{aligned}
d_{sys}(\mathbb{Q}\pi.\psi) &:= 0 \\
d_{sys}(\mathbb{Q}_1\pi.\mathbb{Q}_2\pi'.\varphi) &:= d_{sys}(\mathbb{Q}_2\pi'.\varphi) + q(\mathbb{Q}_1, \mathbb{Q}_2)
\end{aligned}$$

(B) Prefix-cost w.r.t. the size of the *system*. Here, ψ is the quantifier-free body of the formula. The definition of the cost function q is given in Figure 6c.

	\exists	\forall	$\langle\langle A \rangle\rangle$	$\llbracket A \rrbracket$
\exists	0	1	1	1
\forall	1	0	1	1
$\langle\langle A \rangle\rangle$	1	1	2	2
$\llbracket A \rrbracket$	1	1	2	2

(C) Cost associated with each quantifier alteration. Given $\mathbb{Q}_1, \mathbb{Q}_2 \in \{\exists, \forall, \langle\langle A \rangle\rangle, \llbracket A \rrbracket\}$, the cost $q(\mathbb{Q}_1, \mathbb{Q}_2)$ is given in column \mathbb{Q}_1 , row \mathbb{Q}_2 . Note that the cost matrix symmetric.

FIGURE 6. Definition of prefix-cost w.r.t. the size of the specification and the size of the system. Here $\langle\langle A \rangle\rangle$ and $\llbracket A \rrbracket$ represent a *proper* complex quantifier, *i.e.*, a quantifier where $A \neq \emptyset$ and $A \neq \Xi$.

that are *linear*.¹¹ To differentiate formulas based on the structure of their quantifier prefix, we assign each formula φ two quantities: the specification-based prefix-costs $d_{spec}(\varphi)$ and system-based prefix-costs $d_{sys}(\varphi)$. Both are defined inductively in Figures 6a and 6b. Our measures generalize the *alteration-depth* of a HyperLTL formula.¹²

Theorem 6.2. *Model checking of a linear HyperATL^{*} formula φ is*

- (1) *in $d_{spec}(\varphi)$ -EXPTIME in the size of the specification, and*
- (2) *in $d_{sys}(\varphi)$ -EXPTIME in the size of the system*

Proof. Let formula φ and system \mathcal{G} be given. Abbreviate $d_{spec} := d_{spec}(\varphi)$ and $d_{sys} := d_{sys}(\varphi)$. Using the constructions in Figures 4 and 5, we observe that the size of \mathcal{A}_φ is at most $Tower_c(d_{spec}, |\varphi|)$ and $Tower_c(d_{sys}, |\mathcal{G}|)$ for some base c that depends only on d_{spec} and d_{sys} . To argue this, we consider all variations of consecutive types of quantifiers and their cost assigned in Figure 6c. For example, for a formula $\exists\pi.\langle\langle A \rangle\rangle\pi'.\varphi$, the alternating automaton $\mathcal{A}_{\langle\langle A \rangle\rangle\pi'.\varphi}$ needs to be translated to a non-deterministic automaton when performing the construction in Figure 4, which incurs a single exponential blowup. For a formula $\langle\langle A \rangle\rangle\pi.\langle\langle A' \rangle\rangle\pi'.\varphi$, automaton $\mathcal{A}_{\langle\langle A' \rangle\rangle\pi'.\varphi}$ needs to be determinized, causing a double exponential blowup. For a formula $\langle\langle A \rangle\rangle\pi.\exists\pi'.\varphi$, the automaton $\mathcal{A}_{\exists\pi'.\varphi}$ is already nondeterministic, so we can perform a determinization with a single exponential blowup. The cost for all

¹¹To stay as flexible as possible, we include $\llbracket A \rrbracket$ as a first-class quantifier (instead of a derived one). Focusing on linear formulas allows for a simpler characterization of the model checking complexity. We discuss the case of non-linear formulas later in Remark 6.6.

¹²The alteration-depth denotes the number of quantifier alternations (between \exists and \forall) in the quantifier prefix and is used to characterize the HyperLTL model checking complexity [FRS15, Rab16]. For any (linear) HyperATL^{*} formula φ that uses only simple quantification (so φ is also a HyperLTL formula), $d_{sys}(\varphi)$ gives a (in some cases tight) upper bound the alteration-depth of φ . Note that in this case, $d_{spec}(\varphi) = d_{sys}(\varphi) + 1$.

possible combinations of two consecutive quantifier types match with the cost given in Figure 6c. After eliminating all quantifiers, we end up with a alternating automaton over the *singleton alphabet*, for which we can decide emptiness in polynomial time; the bounds follow. Note that $d_{spec}(\varphi)$ and $d_{sys}(\varphi)$ only differ in the cost associated with the innermost quantifier. The automaton construction for this last quantifier is linear in the size of the system but exponential (in case the innermost quantifier is simple) or double exponential (in case the innermost quantifier is complex) in the size of the formula. \square

Remark 6.3. As complex quantification requires a full determinization, simple quantification between complex quantifiers can, in some cases, have no impact on the complexity. For example, for formulas of the form $\langle\langle A \rangle\rangle\pi.\exists\pi'.\langle\langle A' \rangle\rangle\pi''.\psi$ and $\langle\langle A \rangle\rangle\pi.\langle\langle A' \rangle\rangle\pi'.\psi$ our model checking algorithm follows the same asymptotic complexity. \triangleleft

Complexity Based on the Quantifier Type. We obtain a simpler complexity characterization if we only consider the number and type of each quantifier (*i.e.*, we ignore the order in which they occur). We again consider the complexity in the size of the specification (in Proposition 6.4) and the size of the system (in Proposition 6.5).

Proposition 6.4. *Model checking of a linear HyperATL* formula with k complex and l simple quantifiers is*

- (1) *in $(2k + l)$ -EXPTIME, and*
- (2) *in $(2k + l - 1)$ -EXPSPACE if $l \geq 1$*

when measured in the size of the specification.

Proof. Let φ be any formula using k complex and l simple quantifiers, and let \mathcal{G} be the game structure. Point (1) follows directly from Theorem 6.2, as we can easily check that $2k + l \geq d_{spec}(\varphi)$. For point (2) let $\varphi = \mathbb{Q}_1 \pi_1 \dots \mathbb{Q}_{k+l} \pi_{k+l} \cdot \psi$ where ψ is quantifier-free. Let i be the smallest index such that \mathbb{Q}_i is simple (which exists as $l \geq 1$). In case $i = 1$ (*i.e.*, the outermost quantifier is simple) we may assume w.l.o.g that $\mathbb{Q}_1 = \exists$ (as otherwise, we check the negated formula). So \mathcal{A}_φ is a non-deterministic automaton, and the $(2k + l - 1)$ -EXPSPACE upper bound follows, as we can check the emptiness of a non-deterministic automaton in NLOGSPACE [VW94]. If $l > 1$, it is easy to see that $2k + l - 1 \geq d_{spec}(\varphi)$, so we get an even better upper bound of $(2k + l - 1)$ -EXPTIME via Theorem 6.2 and thereby also the desired $(2k + l - 1)$ -EXPSPACE bound. \square

Proposition 6.5. *Model checking of a linear HyperATL* formula with k complex and l simple quantifiers is*

- (1) *in $(2k + l - 2)$ -EXPSPACE, and*
- (2) *in $(2k + l - 2)$ -EXPTIME if $k \geq 1$, and*
- (3) *in $(2k + l - 3)$ -EXPSPACE if $k \geq 1$ and the outermost quantifier is simple (so necessarily $l \geq 1$)*

when measured in the size of the system.

Proof. Let φ be any formula k complex and l simple quantifiers and \mathcal{G} the game structure. We begin with point (2). It is easy to see that if $k \geq 1$, then $2k + l - 2 \geq d_{sys}(\varphi)$, so (2) follows from Theorem 6.2. For point (3) we may assume that the outermost quantifier is existential (otherwise, we check the negated formula). We observe that if $k \geq 1$, the size of \mathcal{A}_φ is at most $Tower_c(2k + l - 2, |\mathcal{G}|)$ (as in the proof for point (2)) and, additionally, \mathcal{A}_φ is non-deterministic, so emptiness can be checked in NLOGSPACE, giving the desired

$(2k + l - 3)$ -EXPSPACE bound. It remains to show point (1). If $k \geq 1$, we get the even better bound of $(2k + l - 2)$ -EXPTIME from point (2). In case $k = 0$ (so all quantifiers are simple), we again assume that the outermost quantifier is existential. It is easy to see that the size of \mathcal{A}_φ is at most $Tower_c(2k + l - 1, |\mathcal{G}|)$ and non-deterministic, so the bound follows from the NLOGSPACE emptiness check. Note that the case where $k = 0$ corresponds directly to HyperLTL model checking [FRS15, Rab16, BF23a]. \square

Remark 6.6. So far, we have focused on linear HyperATL^{*} formulas as this allows for a precise yet succinct analysis. Analogous to point (1) of Proposition 6.4 we can easily see that we can check an arbitrary (possibly non-linear) formula φ with k complex and l simple quantifiers in $(2k + l)$ -EXPTIME in the size of the specification and $(2k + l - 1)$ -EXPTIME in the size of the system (by simply analyzing the size of \mathcal{A}_φ). Deriving more precise bounds by generalizing the specification-based and system-based prefix-cost to non-linear formulas is challenging. \triangleleft

7. LOWER BOUNDS FOR MODEL CHECKING

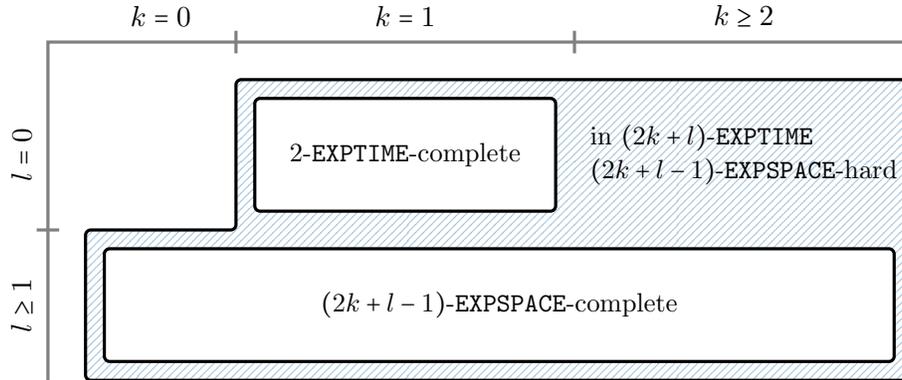
In this section, we establish lower bounds on the HyperATL^{*} model checking problem. Our lower bounds show that strategic quantification in the context of hyperproperties results in a logic that is strictly harder (w.r.t. model checking) than both a hyperlogic without strategic quantification (such as HyperLTL) and a non-hyper logic with strategic quantification (such as ATL^{*}). We establish the following bounds:

Theorem 7.1. *Model checking of a linear HyperATL^{*} formula with k complex and l simple quantifiers is $(2k + l - 1)$ -EXPSPACE-hard in the size of the specification.*

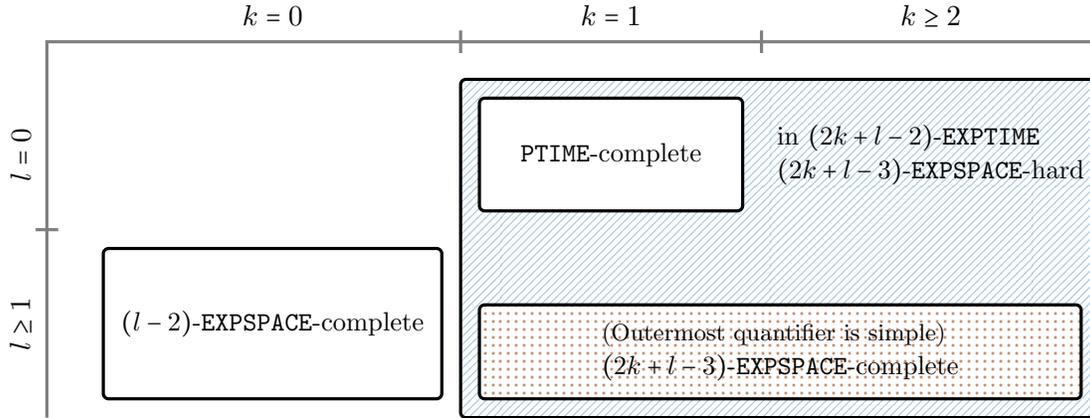
Theorem 7.2. *Model checking of a linear HyperATL^{*} formula with k complex and l simple quantifiers is $(2k + l - 3)$ -EXPSPACE-hard in the size of the system.*

Complexity in the Specification Size. If we consider the HyperATL^{*} model checking complexity in terms of the specification size, Proposition 6.4 and Theorem 7.1 span the landscape depicted in Figure 7a. For all prefix structures, we get an upper bound of $(2k + l)$ -EXPTIME and a lower bound of $(2k + l - 1)$ -EXPSPACE-hardness. In two cases, we can improve the upper or lower bounds further and get tight results: If $k = 1$ and $l = 0$, we get a better 2-EXPTIME lower bound from the ATL^{*} model checking [AHK02] and thus 2-EXPTIME-completeness. In case $l \geq 1$, we get a matching $(2k + l - 1)$ -EXPSPACE upper bound and thus $(2k + l - 1)$ -EXPSPACE-completeness (note that this subsumes the already known HyperLTL bounds in case $k = 0$ [Rab16]).

Complexity in the System Size. If we consider the complexity in the size of the system, Proposition 6.5 and Theorem 7.2 span the landscape depicted in Figure 7b. In case $k = 0$ (where the formula is a HyperLTL formula), we get $(l - 2)$ -EXPSPACE-completeness [Rab16]. In the case where $k \geq 1$ (*i.e.*, the formula is a “proper” HyperATL^{*} formula), we get an upper bound of $(2k + l - 2)$ -EXPTIME and a lower bound of $(2k + l - 3)$ -EXPSPACE-hardness. In two cases we can tighten the results: In case $k = 1$ and $l = 0$, we get a better lower bound from the ATL^{*} model checking and thus PTIME-completeness [AHK02]. In the cases where $l \geq 1$ and the outermost quantifier is simple, we get an improved upper bound resulting in $(2k + l - 3)$ -EXPSPACE-completeness.



(A) Complexity in the size of the specification. $(2k+l)$ -EXPTIME containment and $(2k+l-1)$ -EXPSPACE-hardness holds for the entire fragment (the blue, striped area).



(B) Complexity in the size of the system. $(2k+l-2)$ -EXPTIME containment and $(2k+l-3)$ -EXPSPACE-hardness holds for the blue, striped area. $(2k+l-3)$ -EXPSPACE-completeness only holds for formulas where the outermost quantifier is simple (the red, dotted area).

FIGURE 7. Upper and lower bounds on the complexity of model checking linear HyperATL* formulas with k complex and l simple quantifiers. The bounds are given in the size of the specification (Figure 7a) and size of the system (Figure 7b).

7.1. Proof Preliminaries. The remainder of this section is devoted to a proof of Theorem 7.1 and Theorem 7.2. Readers less interested in a formal proof can skip to Section 8.

Our proof encodes the acceptance of space-bounded Turing machines (TM). It builds on ideas used for the HyperLTL lower bounds shown by Rabe [Rab16] (adopting earlier ideas from Stockmeyer [Sto74]) but uses a novel construction to achieve a doubly exponential increase using complex quantification. The main idea of our construction is to design a HyperATL* formula that requires a player to output a yardstick, which is a formula that specifies a fixed distance between two points along a path. We can then encode the acceptance of a TM by using the yardstick to compare consecutive configurations of the TM. We recommend having a look at the HyperLTL lower bound shown by Rabe [Rab16, §5.6].

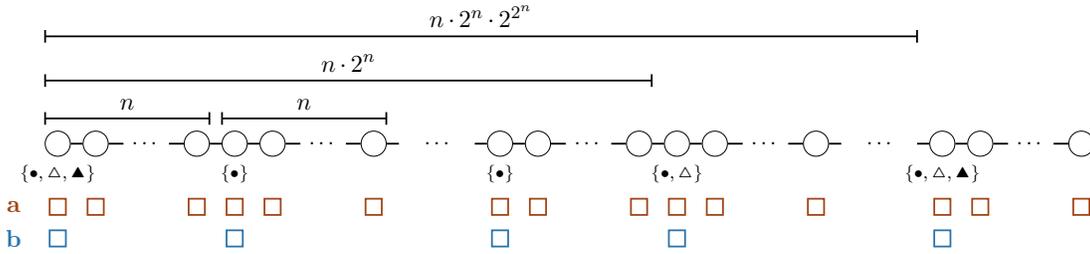


FIGURE 8. The basic structure of a correct counter of length $\mathcal{C}(1, n)$. Proposition \bullet holds every n steps and separates two a -configurations. Proposition Δ holds every $n \cdot 2^n$ step and separates two b -configurations. \blacktriangle holds every $n \cdot 2^n \cdot 2^{2^n}$ steps. Propositions a and b give the bits of a -counter and b -counter, respectively. The a proposition is relevant for the a -counter at all positions, marked by red boxes. The relevant positions for the b -counter are only those where \bullet holds, marked by blue boxes. Consequently, there are 2^n relevant positions for the b -counter between any two occurrences of Δ .

Precise Tower Length. We use a slightly larger tower of exponents. For $k, n \in \mathbb{N}$ define $\mathcal{C}(k, n) \in \mathbb{N}$ as follows:

$$\begin{aligned} \mathcal{C}(0, n) &:= n \\ \mathcal{C}(k+1, n) &:= 2^{2^{\mathcal{C}(k, n)}} \cdot 2^{\mathcal{C}(k, n)} \cdot \mathcal{C}(k, n) \end{aligned}$$

It is easy to see that for every $k \geq 1$, we have $\mathcal{C}(k, n) \geq \text{Tower}_2(2k, n)$ for every n . We design a formula with k complex and 0 simple quantifier that specifies a yardstick of length $\mathcal{C}(k, n)$.¹³ This will later allow us to encode the acceptance of a $\mathcal{C}(k, n)$ -space bounded TM. In our construction, the size of the game structure is constant, and the size of the formula depends on n .

7.2. Direct Counter Verification. We first consider the case where $k = 1$ and construct a formula that ensures a yardstick of length $\mathcal{C}(1, n)$. The idea is to describe a counter with 2^n many bits that is incremented in each step and resets to 0 once the maximal value is reached. Consequently, there are 2^{2^n} counter configurations between two resets of the counter.

Structure of a Counter. To ensure the correctness of the counter with 2^n bits, we use a second counter with n bits. The bits of the larger counter (with 2^n bits) are given via atomic proposition b (called the b -counter), and the bits of the smaller counter (with n bits) by proposition a (called the a -counter). Together, the counter uses atomic propositions $\{\bullet, \Delta, \blacktriangle, a, b\}$, where $\bullet, \Delta, \blacktriangle$ are separating constructs and a, b are propositions that give the value of the counter-bits (if, *e.g.*, proposition a is set, we interpret this as a 1-bit and otherwise as a 0-bit of the a -counter). A *correct* counter has the form depicted in Figure 8. Proposition \bullet occurs every n steps and separates two configurations of the a -counter. The a -counter should continuously count from 0 to $2^n - 1$ (in binary) and then restart at 0 (we use a big-endian encoding where the least significant bit is the last position of a count).

¹³In our proof, we encode the acceptance of $\text{Tower}_2(\cdot, n)$ -space bounded TMs, *i.e.*, we fix the base to 2. Our construction easily extends to an arbitrary (but fixed) base c by using $\lceil \log_2 c \rceil$ propositions for our counter construction. We stick with $c = 2$ to keep the notation simple.

•	■	□	■	□	■	□	■	□	■	□	■	□	■	□	■	□	■		
△	■	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	■	□	□
▲	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
<i>a</i>	□	□	□	■	■	□	■	■	□	□	□	■	■	□	■	■	□	□	□
<i>b</i>	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□
<i>c(a)</i>	0	1	2	3	0	1	2	3	0										
<i>c(b)</i>	0				1														

FIGURE 9. A prefix of a correct counter in the case where $n = 2$. The trace is read from left to right, where each column gives the evaluation of each atomic proposition at that step. ■ means that the proposition holds, and □ that it does not hold. The last two rows give the value of each a -configuration and b -configuration, respectively. Each a -configuration has length 2 ($= n$), and each b -configuration has length 8 ($= n \cdot 2^n$) with 4 ($= 2^n$) relevant bits. The positions where proposition b is irrelevant for the counter are left blank.

The Δ occurs every $n \cdot 2^n$ steps and marks the position where the a -counter resets to 0, *i.e.*, Δ holds whenever the following a -configuration is 0. Proposition b is used for the second counter with 2^n many bits. Each b -configuration is separated by Δ (*i.e.*, one b -configuration has exactly the length in which the a -counter counts from 0 to $2^n - 1$). The relevant bits for the b -counter are only those positions where \bullet holds, so each b -configuration has 2^n relevant bits (marked as blue boxes in Figure 8). The b -counter should count from 0 to $2^{2^n} - 1$ (in binary) and then restart at 0. We mark the reset of the b -counter by \blacktriangle , *i.e.*, \blacktriangle holds whenever the following b -configuration is 0. Consequently, \blacktriangle holds every $\mathcal{C}(1, n) = n \cdot 2^n \cdot 2^{2^n}$ steps. An explicit prefix of a correct counter in the case where $n = 2$ is depicted in Figure 9.

Counter as a Game. We interpret the construction of the counter as a game between a verifier (\mathfrak{V}) and a refuter (\mathfrak{R}). The verifier selects, in each step, the evaluation of the propositions in $\{\bullet, \Delta, \blacktriangle, a, b\}$ that form the counter. Meanwhile, the refuter can challenge the correctness of the counter (we make this precise below). We design a specification and game structure such that the *only* winning strategy for \mathfrak{V} is to produce a correct counter. Consequently, on any play compatible with any winning strategy for \mathfrak{V} , \blacktriangle holds every $\mathcal{C}(1, n)$ steps.

Game Structure. In our game structure, \mathfrak{V} sets the values of the propositions in $\{\bullet, \Delta, \blacktriangle, a, b\}$ and \mathfrak{R} can challenge the correctness by setting proposition *error* and *errorStart*. We allow \mathfrak{R} to postpone the start of the counter. Consider the following game structure over atomic propositions $AP := \{\bullet, \Delta, \blacktriangle, a, b, error, errorStart\}$:

Definition 7.3. Define the CGS $\mathcal{G}_{counter} := (S, s_{init}, \{\mathfrak{V}, \mathfrak{R}\}, \mathcal{M}, \delta, L)$ where

$$S := \{s_O \mid O \subseteq \{\bullet, \Delta, \blacktriangle, a, b, error\}\} \cup \{s_{init}, s_1, s_2, s_3, s_4\}$$

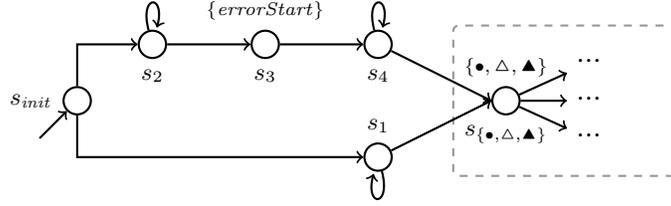


FIGURE 10. The game structure $\mathcal{G}_{counter}$ that is used to produce the counter. The part surrounded by the dashed box, generates the actual counter and includes all states of the form s_O for some $O \subseteq \{\bullet, \Delta, \blacktriangle, a, b, error\}$. The remaining states are used to determine the time point at which the counter should start. The verifier decides the successor whenever in s_4 , and the refuter decides the successor for states s_{init}, s_1, s_2 (state s_3 has a unique successor). We omit the label in case it is empty.

the moves are given by $\mathcal{M} := 2^{\{\bullet, \Delta, \blacktriangle, a, b, error\}} \times \mathbb{B}$. Transition function $\delta : S \times (\{\mathfrak{V}, \mathfrak{R}\} \rightarrow \mathcal{M}) \rightarrow S$ is defined by

$$\delta(s_O, [\mathfrak{V} \mapsto (x, -), \mathfrak{R} \mapsto (y, -)]) := s_{(x \cap \{\bullet, \Delta, \blacktriangle, a, b\}) \cup (y \cap \{error\})}$$

$$\delta(s_{init}, [\mathfrak{V} \mapsto -, \mathfrak{R} \mapsto (-, b)]) := \begin{cases} s_1 & \text{if } b = \top \\ s_2 & \text{if } b = \perp \end{cases}$$

$$\delta(s_1, [\mathfrak{V} \mapsto -, \mathfrak{R} \mapsto (-, b)]) := \begin{cases} s_1 & \text{if } b = \top \\ s_{\{\bullet, \Delta, \blacktriangle\}} & \text{if } b = \perp \end{cases}$$

$$\delta(s_2, [\mathfrak{V} \mapsto -, \mathfrak{R} \mapsto (-, b)]) := \begin{cases} s_2 & \text{if } b = \top \\ s_3 & \text{if } b = \perp \end{cases}$$

$$\delta(s_3, -) := s_4$$

$$\delta(s_4, [\mathfrak{V} \mapsto (-, b), \mathfrak{R} \mapsto -]) := \begin{cases} s_4 & \text{if } b = \top \\ s_{\{\bullet, \Delta, \blacktriangle\}} & \text{if } b = \perp \end{cases}$$

where $-$ denotes an arbitrary value. The labeling $L : S \rightarrow 2^{\{\bullet, \Delta, \blacktriangle, a, b, error\}}$ is defined by

$$L(s_O) = O$$

$$L(s_{init}) = L(s_1) = L(s_2) = L(s_4) = \emptyset$$

$$L(s_3) = \{errorStart\}$$

<

Once a state s_O for some $O \subseteq \{\bullet, \Delta, \blacktriangle, a, b, error\}$ is reached, the verifier can determine which of the propositions in $\{\bullet, \Delta, \blacktriangle, a, b\}$ hold at the next step and refuter decides if *error* holds (the first case in the definition of δ). This part of the state space is responsible for generating the actual counter. The remaining states (s_{init}, s_1, s_2, s_3 , and s_4) are used to determine *when* the counter should start. The structure is sketched in Figure 10. States s_{init}, s_1 , and s_2 are controlled by \mathfrak{R} , *i.e.*, the move selected by \mathfrak{R} determines the successor state, whereas state s_4 is controlled by \mathfrak{V} (see the definition δ). If \mathfrak{R} moves to s_1 , the start of the actual counter can be delayed by looping in s_1 . If \mathfrak{R} moves to s_2 , the *errorStart* proposition occurs at some point, and afterward, \mathfrak{V} can decide when the counter starts by looping in s_4 (this will be of importance to verify the construction in the case $k > 1$).

Specification. We enforce that \mathfrak{V} produces a correct counter once state $s_{\{\bullet, \Delta, \blacktriangle\}}$ is reached for the first time. Consider the HyperATL* specification $correct_1$ defined as follows:

$correct_1$

$$(\neg \blacktriangle_{\pi_1}) \mathcal{W} \left(\blacktriangle_{\pi_1} \wedge \right.$$

$$\quad \square \left(\blacktriangle_{\pi_1} \rightarrow \Delta_{\pi_1} \right) \wedge \left(\Delta_{\pi_1} \rightarrow \bullet_{\pi_1} \right) \wedge \left(\bullet_{\pi_1} \leftrightarrow \bigcirc^n \bullet_{\pi_1} \right) \wedge \quad (7.1)$$

$$\quad \square \left(\Delta_{\pi_1} \leftrightarrow \left(\neg a_{\pi_1} \wedge \bigcirc \left(\left(\neg a_{\pi_1} \right) \mathcal{U} \bullet_{\pi_1} \right) \right) \right) \wedge \quad (7.2)$$

$$\quad \square \left(\blacktriangle_{\pi_1} \leftrightarrow \left(\neg b_{\pi_1} \wedge \bigcirc \left(\left(\bullet_{\pi_1} \rightarrow \neg b_{\pi_1} \right) \mathcal{U} \Delta_{\pi_1} \right) \right) \right) \wedge \quad (7.3)$$

$$\quad \square \left(\left(a_{\pi_1} \leftrightarrow \bigcirc^n a_{\pi_1} \right) \leftrightarrow \bigcirc \text{before} \left(\neg a_{\pi_1}, \bullet_{\pi_1} \right) \right) \wedge \quad (7.4)$$

$$\quad \left. \left(\left(\text{exactlyOnce} \left(\text{error}_{\pi_1} \right) \wedge \square \left(\text{error}_{\pi_1} \rightarrow \bullet_{\pi_1} \right) \right) \rightarrow \text{falseAlarm}_1 \right) \right) \quad (7.5)$$

The initial weak until accounts for the offset before the counter is started, *i.e.*, the counter should be correct once \blacktriangle_{π_1} holds for the first time (which is the case when state $s_{\{\bullet, \Delta, \blacktriangle\}}$ is reached for the first time). We discuss the conjuncts 7.1 - 7.5 in detail:

- (7.1) Formula 7.1 states that the basic behavior of the separating constructs is correct: The implications between \blacktriangle , Δ and \bullet hold, and \bullet occurs every n steps.
- (7.2) Formula 7.2, ensures that the Δ proposition is set correctly. Δ should hold exactly if the next a -configuration is 0, *i.e.*, a does not hold until the next a -configuration begins (which is marked by \bullet).
- (7.3) Similar to Formula 7.2, Formula 7.3 ensures that \blacktriangle is set exactly if the next b -configuration is 0, so b does not hold until the next b -configuration begins (marked by Δ). Recall that the bits of the b -counter are only those where \bullet holds.
- (7.4) Formula 7.4 ensures the correctness of the a -counter. Here

$$\text{before}(\varphi, \psi) := (\neg \psi) \mathcal{U} (\varphi \wedge \neg \psi)$$

expresses that φ should hold at some time strictly before ψ holds for the first time. To encode that the a -counter is incremented (or reset), we use the following fact. Assume we are given two m -bit counters $\alpha = \alpha_0, \dots, \alpha_m$ and $\beta = \beta_0, \dots, \beta_m$ (with big-endian encoding, *i.e.*, α_m, β_m are the least significant bits). Let $c(\alpha), c(\beta) \in \{0, \dots, 2^m - 1\}$ give the value of the counters. Then

$$\left[c(\beta) = c(\alpha) + 1 \pmod{2^m} \right] \text{ iff } \left[\forall i. (\alpha_i = \beta_i) \leftrightarrow (\exists j > i. \alpha_j = 0) \right]. \quad (7.6)$$

Formula 7.4 thus encodes that the value of the a -counter is incremented by using \bigcirc^n to compare the same position across two consecutive a -configurations.

- (7.5) Lastly, Formula 7.5 ensures the correctness of the b -counter. Here

$$\text{exactlyOnce}(\psi) := (\neg \psi) \mathcal{U} (\psi \wedge \bigcirc \square (\neg \psi))$$

expresses that ψ holds exactly once. Different from Formula 7.3, we cannot encode the correctness directly via \bigcirc s as the relevant positions of the b -counter are exponentially many steps apart. Instead, we use \mathfrak{R} 's ability to raise the *error* proposition. To challenge the correctness, \mathfrak{R} should set *error* at the *first* bit in the b -counter that is incorrect. After the challenge occurred, we check if \mathfrak{R} spotted a genuine mistake in the

counter (so $falseAlarm_1$ should state that the challenge was a false alarm; the counter at the position pointed to by $error$ was correct). To check the challenge we need to compare the position marked with $error$ with the same position in the *previous* b -configuration (which is the position $n \cdot 2^n$ steps earlier). The crux is that once the $error$ proposition is set, we can identify this position by using the a -counter (which we can assume to be correct). Consider the formula $prevPos$ defined as follows:

$prevPos$

$$\bullet_{\pi_1} \wedge \quad (7.7)$$

$$\left((\neg \Delta_{\pi_1}) \mathcal{U} \left(\Delta_{\pi_1} \wedge \bigcirc \left((\neg \Delta_{\pi_1}) \mathcal{U} error_{\pi_1} \right) \right) \right) \wedge \quad (7.8)$$

$$\left(\bigwedge_{i=0}^{n-1} \left(\bigcirc^i a_{\pi_1} \leftrightarrow \square \left(error_{\pi_1} \rightarrow \bigcirc^i a_{\pi_1} \right) \right) \right) \quad (7.9)$$

Formula $prevPos$ holds exactly once (assuming that the a -counter is correct and $error_{\pi_1}$ occurs exactly once), and identifies the same position in the b -configuration that precedes that in which $error_{\pi_1}$ holds. There are three conditions that identify this (unique) position:

(7.7) The position aligns with \bullet_{π_1} (*i.e.*, a position where b is relevant).

(7.8) The position lies within the b -configuration that precedes the b -configuration in which $error_{\pi_1}$ holds, *i.e.*, Δ_{π_1} holds exactly once before $error_{\pi_1}$ holds.

(7.9) The position corresponds to the same bit of the b -counter. Each bit of the b -counter is uniquely characterized by the a -configuration that starts at that bit. To check that we are at the same position, the a -configuration should thus be the same as the a -configuration at the position pointed to by $error$.

Using $prevPos$, we can then state that $error$ does not mark a genuine error in the b -counter. Here we again make use of Equation (7.6). We define formula $falseAlarm_1$ (which is used in formula $correct_1$) as follows:

$falseAlarm_1$

$$\left(\square \left(prevPos \rightarrow b_{\pi_1} \right) \leftrightarrow \square \left(error_{\pi_1} \rightarrow b_{\pi_1} \right) \right) \\ \leftrightarrow \left(\square \left(prevPos \rightarrow \bigcirc before \left(\bullet_{\pi_1} \wedge \neg b_{\pi_1}, \Delta_{\pi_1} \right) \right) \right)$$

The bit of the b -counter at the position where $prevPos$ and the bit where $error_{\pi_1}$ holds (note that both positions are unique) should agree iff in the b -configuration pointed to by $prevPos$ there exists a 0-bit at a less significant position (*cf.* Equation (7.6)).

It is easy to see that $\langle\langle \mathfrak{V} \rangle\rangle_{\pi_1}. correct_1$ holds on $\mathcal{G}_{counter}$. The only winning strategy for \mathfrak{V} is to output a correct counter (as soon as state $s_{\{\bullet, \Delta, \blacktriangle\}}$ is reached). In particular, on any play produced by a winning strategy for \mathfrak{V} , \blacktriangle_{π_1} occurs exactly every $\mathcal{C}(1, n)$ steps (once $s_{\{\bullet, \Delta, \blacktriangle\}}$ is reached for the first time).

7.3. Counter Verification Using Smaller Counter. To obtain a yardstick of length $\mathcal{C}(k, n)$ for $k > 1$, we use the same counter structure in Figure 8 but set the length (number of bits) of the a -counter to be $\mathcal{C}(k - 1, n)$. To verify the correctness of the counter, we then

use a smaller yardstick of length $\mathcal{C}(k-1, n)$. The final formula has the form

$$\langle\langle \mathfrak{V} \rangle\rangle_{\pi_k} \dots \langle\langle \mathfrak{V} \rangle\rangle_{\pi_1} \wedge \bigwedge_{i=1}^k \text{correct}_i.$$

We assert that every winning strategy for \mathfrak{V} constructing path π_i encodes a counter of length $\mathcal{C}(i, n)$ and use the counter on π_{i-1} (which we can inductively assume to be correct) to ensure its correctness. The construction in Section 7.2 gives the base case for $k = 1$. To verify the correctness of the counter on π_i (for $i > 1$), we make use of the two modes available to \mathfrak{R} in $\mathcal{G}_{\text{counter}}$ (see Figure 10). By moving to state s_1 , \mathfrak{R} can start the counter at any time (we will use this to verify that the placement of \bullet and the a -counter is correct). By moving to s_2 , \mathfrak{R} can set the *errorStart* proposition at any time, after which \mathfrak{V} can postpone the start of the counter. We use this to verify the correctness of the b -counter.

For $i > 1$, define formula correct_i as follows:

correct_i

$$\begin{aligned}
& (\neg \blacktriangle_{\pi_i}) \mathcal{W} \left(\blacktriangle_{\pi_i} \wedge \right. \\
& \quad \square \left((\blacktriangle_{\pi_i} \rightarrow \triangle_{\pi_i}) \wedge (\triangle_{\pi_i} \rightarrow \bullet_{\pi_i}) \right) \wedge \tag{7.10} \\
& \quad \square \left(\blacktriangle_{\pi_{i-1}} \wedge \bullet_{\pi_i} \rightarrow (\neg \bullet_{\pi_i} \wedge \neg \blacktriangle_{\pi_{i-1}}) \mathcal{U} (\bullet_{\pi_i} \wedge \blacktriangle_{\pi_{i-1}}) \right) \wedge \tag{7.11} \\
& \quad \square \left(\triangle_{\pi_i} \leftrightarrow (\neg a_{\pi_i} \wedge \bigcirc ((\neg a_{\pi_i}) \mathcal{U} \bullet_{\pi_i})) \right) \wedge \tag{7.12} \\
& \quad \square \left(\blacktriangle_{\pi_i} \leftrightarrow (\neg b_{\pi_i} \wedge \bigcirc ((\bullet_{\pi_i} \rightarrow \neg b_{\pi_i}) \mathcal{U} \triangle_{\pi_i})) \right) \wedge \tag{7.13} \\
& \quad \square \left((a_{\pi_i} \wedge \blacktriangle_{\pi_{i-1}} \wedge \text{next}(\blacktriangle_{\pi_{i-1}}, a_{\pi_i})) \rightarrow \bigcirc \text{before}(\neg a_{\pi_i}, \bullet_{\pi_i}) \right) \wedge \\
& \quad \square \left((\neg a_{\pi_i} \wedge \blacktriangle_{\pi_{i-1}} \wedge \text{next}(\blacktriangle_{\pi_{i-1}}, \neg a_{\pi_i})) \rightarrow \bigcirc \text{before}(\neg a_{\pi_i}, \bullet_{\pi_i}) \right) \wedge \tag{7.14} \\
& \quad \square \left((\neg a_{\pi_i} \wedge \blacktriangle_{\pi_{i-1}} \wedge \text{next}(\blacktriangle_{\pi_{i-1}}, a_{\pi_i})) \rightarrow \bigcirc (a_{\pi_i} \mathcal{U} \bullet_{\pi_i}) \right) \wedge \\
& \quad \square \left((a_{\pi_i} \wedge \blacktriangle_{\pi_{i-1}} \wedge \text{next}(\blacktriangle_{\pi_{i-1}}, \neg a_{\pi_i})) \rightarrow \bigcirc (a_{\pi_i} \mathcal{U} \bullet_{\pi_i}) \right) \wedge \\
& \quad \left. \left(\text{exactlyOnce}(\text{error}_{\pi_i}) \wedge \text{exactlyOnce}(\text{errorStart}_{\pi_{i-1}}) \rightarrow \text{falseAlarm}_i \right) \right) \tag{7.15}
\end{aligned}$$

We again discuss each conjunct separately. Let $\tilde{n} := \mathcal{C}(i-1, n)$ be the length of the smaller yardstick on π_{i-1}

(7.10) Formula 7.10 ensures the basic implications between \bullet , \triangle and \blacktriangle .

(7.11) Formula 7.11 ensures that \bullet_{π_i} holds every \tilde{n} steps. Different from Formula 7.1 we cannot express this using $\bigcirc \mathfrak{s}$. Instead, we use the smaller counter that will be generated on π_{i-1} . On π_{i-1} , \mathfrak{R} can start the counter at any possible time (by looping in s_1). Formula 7.11 now states that if $\blacktriangle_{\pi_{i-1}}$ holds at the same time as \bullet_{π_i} , then \bullet_{π_i} should hold the next time $\blacktriangle_{\pi_{i-1}}$ holds (which is \tilde{n} steps apart if the counter on π_{i-1} is correct). As \mathfrak{R} can start the counter on π_{i-1} at any time and π_{i-1} is resolved *after* π_i is fixed, any winning strategy for \mathfrak{V} on π_i must set \bullet_{π_i} exactly \tilde{n} steps apart.

(7.12) Formula 7.12 ensures that \triangle_{π_i} holds iff the next a -configuration is 0.

(7.13) Formula 7.13 ensures that \blacktriangle_{π_i} holds iff the next b -configuration is 0.

(7.14) Formula 7.14 ensures the correctness of the a -counter. We split this into 4 separate conditions, and, similar to Formula 7.11, use the counter on π_{i-1} to compare positions

that are \tilde{n} steps apart. Here

$$\text{next}(\varphi, \psi) := (-\varphi)\mathcal{U}(\varphi \wedge \psi)$$

expresses that ψ should hold at the next occurrence of φ . For example, the first line of Formula 7.14 covers the following case: If both a_{π_i} and $\blacktriangle_{\pi_{i-1}}$ hold now and the next time $\blacktriangle_{\pi_{i-1}}$ holds, a_{π_i} also holds (so the current bit of the current a -configuration does not change), then there should be a 0-bit before the end of the current a -configuration (cf. Equation (7.6)). Similarly, if the value of a_{π_i} now and the next time $\blacktriangle_{\pi_{i-1}}$ holds is different, then there should not be a 0-bit before the end of the current a -configuration (expressed in the last two cases of Formula 7.14). As \mathfrak{A} can start the counter on π_{i-1} at any time, any bits in consecutive a -configurations on π_i can be compared, so the only winning strategy for \mathfrak{V} on π_i is to produce a correct a -counter.

- (7.15) Ensuring the correctness of the b -counter is more challenging. We again let \mathfrak{A} challenge the correctness of the b -counter by setting error . However, different from Formula 7.5, we cannot directly identify the same position in the previous b -configuration (note that the construction of prevPos depends on n). Instead, if error_{π_i} is set by \mathfrak{A} , \mathfrak{A} is also responsible for identifying the same position in the *previous* b -configuration by setting $\text{errorStart}_{\pi_{i-1}}$ on π_{i-1} (which he can do by moving the game producing π_{i-1} to state s_2 , see Figure 10). We can then compare the two bits of the b -counter pointed to by $\text{errorStart}_{\pi_{i-1}}$ and error_{π_i} . As the position where $\text{errorStart}_{\pi_{i-1}}$ is set is determined by \mathfrak{A} we additionally need to check that the position is correct, *i.e.*, corresponds to the same position within the previous b -configuration. We define formula falseAlarm_i as follows:

falseAlarm_i

$$\text{counterIsCorrect}_i \vee \text{wrongConfiguration}_i \vee \text{wrongBit}_i$$

We discuss each disjunct of falseAlarm_i separately. Note that only one of these disjuncts needs to hold in order to show that the supposed error identified by \mathfrak{A} is not genuine.

- $\text{counterIsCorrect}_i$ expresses that the two positions pointed to by $\text{errorStart}_{\pi_{i-1}}$ and error_{π_i} are correct, *i.e.*, \mathfrak{A} did not point to an actual error in the b -counter. We define it as follows:

$\text{counterIsCorrect}_i$

$$\begin{aligned} & \left(\Box(\text{errorStart}_{\pi_{i-1}} \rightarrow b_{\pi_i}) \leftrightarrow \Box(\text{error}_{\pi_i} \rightarrow b_{\pi_i}) \right) \\ & \leftrightarrow \left(\Box(\text{errorStart}_{\pi_{i-1}} \rightarrow \bigcirc \text{before}(\bullet_{\pi_i} \wedge \neg b_{\pi_i}, \Delta_{\pi_i})) \right) \end{aligned}$$

The formula is similar to falseAlarm_1 , but uses $\text{errorStart}_{\pi_{i-1}}$ instead of prevPos to point to the same position in the previous b -configuration.

- $\text{wrongConfiguration}_i$ expresses that $\text{errorStart}_{\pi_{i-1}}$ and error_{π_i} do not occur in two consecutive b -configurations on π_i , *i.e.*, there is not exactly one Δ_{π_i} between both.

$wrongConfiguration_i$

$$\Box \left(errorStart_{\pi_{i-1}} \rightarrow \neg \left((\neg \Delta_{\pi_i}) \mathcal{U} \left(\Delta_{\pi_i} \wedge \bigcirc \left((\neg \Delta_{\pi_i}) \mathcal{U} error_{\pi_i} \right) \right) \right) \right)$$

- $wrongBit_i$ expresses that $errorStart_{\pi_{i-1}}$ and $error_{\pi_i}$ do not point to the same bit in the two consecutive b -configurations. We again use the fact that a bit of the b -counter is precisely characterized by the a -configuration that starts at the bit. If \mathfrak{R} moved the game producing π_{i-1} to s_3 (which he did as $errorStart_{\pi_{i-1}}$ occurs), \mathfrak{V} can loop in state s_4 and decide when to start the counter. To show that $errorStart_{\pi_{i-1}}$ and $error_{\pi_i}$ point to different b -bits on π_i , \mathfrak{V} should loop in s_4 and find a bit position at which the a -configuration that starts at position $errorStart_{\pi_{i-1}}$ and the a -configuration that starts at position $error_{\pi_i}$ differ. We define $wrongBit_i$ as follows:

$wrongBit_i$

$$\Box \left(errorStart_{\pi_{i-1}} \rightarrow before(\blacktriangle_{\pi_{i-1}}, \bullet_{\pi_i}) \right) \wedge \tag{7.16}$$

$$\neg \left(\left(\Box \left(errorStart_{\pi_{i-1}} \rightarrow next(\blacktriangle_{\pi_{i-1}}, a_{\pi_i}) \right) \right) \right) \tag{7.17}$$

$$\leftrightarrow \left(\Box \left(error_{\pi_i} \rightarrow next(\blacktriangle_{\pi_{i-1}}, a_{\pi_i}) \right) \right)$$

Formula 7.16 ensures that \mathfrak{V} starts the counter soon enough by leaving s_4 , *i.e.*, after $errorStart_{\pi_{i-1}}$ is set, the counter on π_{i-1} is started within the same a -configuration on π_i (before \bullet_{π_i} holds). Formula 7.17 states that \mathfrak{V} started the counter at a time that shows that \mathfrak{R} set $errorStart_{\pi_{i-1}}$ at a wrong location (*i.e.*, did not point to a genuine error). That is, the bit of the a -configuration where $\blacktriangle_{\pi_{i-1}}$ holds (for the first time) after $errorStart_{\pi_{i-1}}$ differs from the value of the a -configuration where $\blacktriangle_{\pi_{i-1}}$ holds (for the first time) after $error_{\pi_i}$.

7.4. Lower Bound Proofs. We use the counter construction to prove Theorem 7.1 and Theorem 7.2.

Theorem 7.1. *Model checking of a linear HyperATL* formula with k complex and l simple quantifiers is $(2k + l - 1)$ -EXPSpace-hard in the size of the specification.*

Proof. In the case where $k = 0$, the HyperATL* formula is a HyperLTL formula and we can reuse the $(l - 1)$ -EXPSpace lower bound shown by Rabe [Rab16]. So let us assume that $k \geq 1$. We distinguish if $l \geq 1$ or $l = 0$.

- If $l \geq 1$: The counter construction in Section 7.2 and Section 7.3 gives us a formula of the form

$$\langle\langle \mathfrak{V} \rangle\rangle \pi_k \dots \langle\langle \mathfrak{V} \rangle\rangle \pi_1 \cdot \bigwedge_{i=1}^k correct_i$$

and a game structure $\mathcal{G}_{counter}$ (note that the size of $\mathcal{G}_{counter}$ is independent of n), such that \mathfrak{R} can start the counter on path π_k at any time and, once started, \mathfrak{V} needs to produce a correct counter, *i.e.*, \blacktriangle_{π_k} holds every $\mathcal{C}(k, n)$ steps. Given a $\mathcal{C}(k, n)$ -space-bounded Turing

machine \mathcal{T} and an input w (with $|w| = n$), we can design a formula of the form

$$\exists \pi. \langle\langle \mathfrak{A} \rangle\rangle \pi_k \dots \langle\langle \mathfrak{A} \rangle\rangle \pi_1. \psi_{(\mathcal{T}, w)} \quad (7.18)$$

and a game structure $\mathcal{G}_{\mathcal{T}}$ (the size of which is constant and does not depend on w), such that $\mathcal{G}_{\mathcal{T}} \models (7.18)$ iff \mathcal{T} accepts w . The idea of this encoding is similar to [Rab16]: First, formula $\psi_{(\mathcal{T}, w)}$ contains $\bigwedge_{i=1}^k \text{correct}_i$ as a conjunct to ensure that the counters on paths π_1, \dots, π_k are correct. In addition, the path π should enumerate consecutive configurations of \mathcal{T} (each of which is $\mathcal{C}(k, n)$ steps long). The initial configuration should contain the input w (which we simply hard-code in the formula). Using the yardstick (which \mathfrak{A} can start at any time), we can compare positions which are $\mathcal{C}(k, n)$ -steps apart and – as transitions of a TM are local – enforce that π encodes a valid accepting computation (see [Rab16, Lemma 5.6.3] for details). Model checking of a formula of the form (7.18) (with k complex and 1 simple quantifier) is thus $2k$ -EXPSPACE-hard (Note that $\mathcal{C}(k, n) \geq \text{Tower}_2(2k, n)$. We can scale the counter to an arbitrary base $c > 2$ by using $\lceil \log_2 c \rceil$ propositions for the counter). In cases of more than a single simple quantifier, we can construct a larger yardstick by adding the construction of Rabe [Rab16] (which extends the length by one exponent with each simple quantifier) to ours (which extends the length by two exponents with each complex quantifier). See [Rab16, Lemma 5.6.2] for details.

- If $l = 0$: Similar to the previous case, we use our counter construction. Assume we are given a $2^{\mathcal{C}(k-1, n)}$ -space-bounded Turing machine \mathcal{T} and an input w (with $|w| = n$). We design a formula

$$\langle\langle \mathfrak{A} \rangle\rangle \pi. \langle\langle \mathfrak{A} \rangle\rangle \pi_{k-1} \dots \langle\langle \mathfrak{A} \rangle\rangle \pi_1. \psi_{(\mathcal{T}, w)} \quad (7.19)$$

and game structure such that \mathfrak{A} on path π should produce a correct counter with $\mathcal{C}(k-1, n)$ many bits (similar to the a -counter) and in place of the b -counter output configurations of \mathcal{T} (so each configuration has length $2^{\mathcal{C}(k-1, n)}$). We use the yardstick of length $\mathcal{C}(k-1, n)$ on π_{k-1} to verify the correctness of the a -counter on π (if $k = 1$, we verify it directly using Os). We verify that consecutive configurations of \mathcal{T} are correct similar to the verification of the b -counter via the *error* proposition (which now points to errors in the TM configurations as opposed to errors in the b -counter). Model checking a formula with k complex quantifiers is thus $(2k - 1)$ -EXPSPACE-hard. \square

Theorem 7.2. *Model checking of a linear HyperATL^{*} formula with k complex and l simple quantifiers is $(2k + l - 3)$ -EXPSPACE-hard in the size of the system.*

Proof. If $k = 0$, we get an $(l - 2)$ -EXPSPACE lower bound from HyperLTL model checking hardness [Rab16] which is even better than the $(l - 3)$ -EXPSPACE bound required. So let us assume that $k \geq 1$. We distinguish if $l \geq 1$ or $l = 0$.

- If $l \geq 1$: We first observe that we can construct a formula of the form

$$\langle\langle \mathfrak{A} \rangle\rangle \pi_k \dots \langle\langle \mathfrak{A} \rangle\rangle \pi_1. \bigwedge_{i=1}^k \text{correct}'_i$$

of constant size and a CGS \mathcal{G} (whose size depends on n) such that \mathfrak{A} is required to output a yardstick of length $\mathcal{C}(k-1, n)$ on π_k . The construction is similar to the counter defined in Section 7.2 and Section 7.3 but modifies correct_1 (note that the size of correct_1 depends on n). We ensure that π_1 no longer produces a yardstick of length $\mathcal{C}(1, n)$ but only of length n . We modify the game structure such that n is hard-coded (*i.e.*, \blacktriangle occurs exactly every n steps) and can ensure the correctness of the n -bit counter between two \blacktriangle s with

a formula that does not depend on n . Similar to Theorem 7.1, we can then encode the acceptance of a $\mathcal{C}(k-1, n)$ -space bounded TM \mathcal{T} on input w (with $|w| = n$) as a formula

$$\exists \pi. \langle\langle \mathfrak{A} \rangle\rangle \pi_k \dots \langle\langle \mathfrak{A} \rangle\rangle \pi_1. \psi_{(\mathcal{T}, w)}$$

whose size does not depend on the size of input w (the input is hard-coded in the game structure). Verification of a formula of the above form (with k complex and 1 simple quantifier) is thus $(2k-2)$ -EXPSpace-hard in the size of the system. In case of more than a single simple quantifier, we, again, use the construction of Rabe [Rab16] to extend the yardstick.

- If $l = 0$: This is analogous to the second case in the proof of Theorem 7.1. To ensure that the size of the formula is independent of n , we again let π_1 only produce a counter of length n (compared to $\mathcal{C}(1, n)$ in the proof of Theorem 7.1). \square

8. EXPERIMENTAL EVALUATION

As indicated by our lower bounds, **HyperATL*** model checking for the full logic is not practical. Instead, we focus on formulas of the form $[\langle\langle A_1 \rangle\rangle \pi_1 \dots \langle\langle A_n \rangle\rangle \pi_n] \psi$ where ψ is quantifier-free. In terms of model checking complexity, this fragment is much cheaper than full **HyperATL***; it is 2-EXPTIME-complete in the size of the specification (as the fragment still subsumes LTL realizability) and PTIME-complete in the size of the system. This fragment of **HyperATL*** is expressive: It subsumes alternation-free **HyperLTL** specifications, many security specifications such as simulation-based security (Section 4.3), the game-based model checking approximation from [CFST19] (Section 4.2), and the asynchronous approach from Section 5.4. Model checking a formula in the above fragment can be reduced to the solving of a parity game by building the product of the game structure with a deterministic parity automaton for ψ . We account for the order of move selection by simulating a single step in the CGS with multiple intermediate steps where all agents in the same stage fix their move incrementally.

We have implemented this construction in **hyperatlmc**, a prototype model checker for **HyperATL*** formulas of the form $[\langle\langle A_1 \rangle\rangle \pi_1 \dots \langle\langle A_n \rangle\rangle \pi_n] \psi$ where ψ is quantifier-free. Our tool uses **rabinizer4** [KMSZ18] to compute deterministic parity automata and **pgsolver** [FL09] to solve parity games.

In this section, we give a simple operational semantics for a boolean programming language into CGSs (in Section 8.1). This allows us to check the (synchronous and asynchronous) security properties from Sections 4 and 5 (which are stated at the level of CGSs) on programs. Afterward, we report on experiments with **hyperatlmc** in Section 8.2.

8.1. Compiling Programs into Game Structures. To have a fixed language to express programs, we use a simple toy-language we call **bwhile**. We endow **bwhile** programs with a direct semantics into a game structures over players ξ_L, ξ_H , and ξ_N (*cf.* Section 4), which allows us to apply the properties given in Sections 4 and 5.

We fix a finite set of program variables \mathcal{X} and define boolean expression as follows:

$$e := x \mid true \mid false \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg e$$

where $x \in \mathcal{X}$.

$$\begin{array}{c}
\frac{}{\langle x \leftarrow e, \mu \rangle \rightsquigarrow \langle b, \mu[x \mapsto \llbracket e \rrbracket(\mu)] \rangle} \quad \frac{\llbracket e \rrbracket(\mu) = \top}{\langle \text{if}(e, P_1, P_2), \mu \rangle \rightsquigarrow \langle P_1, \mu \rangle} \quad \frac{\llbracket e \rrbracket(\mu) = \perp}{\langle \text{if}(e, P_1, P_2), \mu \rangle \rightsquigarrow \langle P_2, \mu \rangle} \\
\\
\frac{b \in \mathbb{B} \quad P \in \{L, H\}}{\langle x \leftarrow \text{Read}_P, \mu \rangle \rightsquigarrow \langle b, \mu[x \mapsto b] \rangle} \quad \frac{}{\langle P_1 \oplus P_2, \mu \rangle \rightsquigarrow \langle P_1, \mu \rangle} \quad \frac{}{\langle P_1 \oplus P_2, \mu \rangle \rightsquigarrow \langle P_2, \mu \rangle} \\
\\
\frac{\llbracket e \rrbracket(\mu) = \perp}{\langle \text{while}(e, P), \mu \rangle \rightsquigarrow \langle b, \mu \rangle} \quad \frac{\llbracket e \rrbracket(\mu) = \top}{\langle \text{while}(e, P), \mu \rangle \rightsquigarrow \langle P; \text{while}(e, P), \mu \rangle} \\
\\
\frac{\langle P_1, \mu \rangle \rightsquigarrow \langle b, \mu' \rangle}{\langle P_1; P_2, \mu \rangle \rightsquigarrow \langle P_2, \mu' \rangle} \quad \frac{\langle P_1, \mu \rangle \rightsquigarrow \langle P'_1, \mu' \rangle \quad P'_1 \neq b}{\langle P_1; P_2, \mu \rangle \rightsquigarrow \langle P'_1; P_2, \mu' \rangle} \quad \frac{}{\langle b, \mu \rangle \rightsquigarrow \langle b, \mu \rangle}
\end{array}$$

FIGURE 11. Small-step semantics for **while**. A step has the form $\langle P, \mu \rangle \rightsquigarrow \langle P', \mu' \rangle$ where program P in memory μ steps to program P' and memory μ' . For a boolean expression e and memory μ we write $\llbracket e \rrbracket(\mu) \in \mathbb{B}$ for the value of e in μ (defined as expected).

while programs are then generated by the following grammar:

$$P := x \leftarrow e \mid x \leftarrow \text{Read}_H \mid x \leftarrow \text{Read}_L \mid \text{if}(e, P_1, P_2) \mid P_1 \oplus P_2 \mid \text{while}(e, P) \mid P_1; P_2 \mid b$$

where $x \in \mathcal{X}$. Most language constructs are standard: $x \leftarrow \text{Read}_H$ (resp. $x \leftarrow \text{Read}_L$) reads the value of x from a high-security (resp. low security) source, $P_1 \oplus P_2$ is a nondeterministic choice between P_1 and P_2 and b is the terminated program. We endow our language with a standard small-step semantics operating on configurations of the form $\langle P, \mu \rangle$ where P is a program and $\mu : \mathcal{X} \rightarrow \mathbb{B}$ a memory. The reduction steps are standard; we give them in Figure 11 for completeness. To obtain a game structure, we associated each program P to a player $\langle P \rangle \in \{\xi_N, \xi_H, \xi_L\}$, where player $\langle P \rangle$ decides on the successor state of P . We define $\langle P \rangle$ as follows:

$$\begin{array}{llll}
\langle x \leftarrow \text{Read}_L \rangle := \xi_L & \langle x \leftarrow \text{Read}_H \rangle := \xi_H & \langle P_1 \oplus P_2 \rangle := \xi_N & \langle P_1; P_2 \rangle := \langle P_1 \rangle \\
\langle x \leftarrow e \rangle := \xi_N & \langle \text{if}(e, P_1, P_2) \rangle := \xi_N & \langle \text{while}(e, P) \rangle := \xi_N & \langle b \rangle := \xi_N
\end{array}$$

Given a program \dot{P} , we define the game structure $\mathcal{G}_{\dot{P}}$ over agents $\{\xi_N, \xi_H, \xi_L\}$ as follows: The states of $\mathcal{G}_{\dot{P}}$ are all configurations $\langle P, \mu \rangle$ where P is a program and μ a memory. The initial state is $\langle \dot{P}, \lambda_{\perp} \rangle$ (*i.e.*, the initial memory assigns all variables to \perp). In state $\langle P, \mu \rangle$, player $\langle P \rangle$ decides on a successor state from the set $\{\langle P', \mu' \rangle \mid \langle P, \mu \rangle \rightsquigarrow \langle P', \mu' \rangle\}$.¹⁴ $\mathcal{G}_{\dot{P}}$ is a turn-based game in the sense of [AHK02]. Note that the state-space of $\mathcal{G}_{\dot{P}}$ is infinite (as there are infinitely many programs), but the reachable fragment is finite and computable. The atomic propositions in $\mathcal{G}_{\dot{P}}$ are all variables from \mathcal{X} that are used in \dot{P} . An atomic proposition (variable) $x \in \mathcal{X}$ holds in state $\langle P, \mu \rangle$ iff $\mu(x) = \top$.

8.2. Experiments. We applied `hyperatlmc` to small **while** programs and checked synchronous and asynchronous information flow policies.

¹⁴Note that for all constructs except $P_1 \oplus P_2$, $x \leftarrow \text{Read}_L$, $x \leftarrow \text{Read}_H$, and $P_1; P_2$ there is unique successor configuration (so the player is irrelevant). As expected, ξ_L chooses the successor of a program $x \leftarrow \text{Read}_L$ and thereby fixes the next value of x (and similarly for ξ_H and $x \leftarrow \text{Read}_H$). In a non-deterministic branching $P_1 \oplus P_2$, player ξ_N decides which branch to take.

<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">P1</div> <pre> o ← true while(true) o ← ¬o </pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">P3</div> <pre> o ← true while(true) l ← Read_L if * then o ← l else o ← ¬l </pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">P4</div> <pre> o ← true while(true) if * then h ← Read_H o ← h else h ← Read_H o ← ¬h </pre>
<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">P2</div> <pre> o ← true while(true) l ← Read_L o ← l </pre>		

FIGURE 12. Simple `bwhile` example programs that distinguish different information-flow policies. We write `if * then P1 else P2` instead of $P_1 \oplus P_2$.

TABLE 1. Model checking results for information-flow properties (expressed in HyperATL*) on the small `bwhile` programs from Figure 12. We give the model checking result (Res) (✓ indicates that the property holds, and ✗ that it is violated) and time taken by `hyperatlmc` in milliseconds (t).

Program	(OD)		(NI)		(simSec)		(approxGNI ₃)	
	Res	t	Res	t	Res	t	Res	t
P1	✓	12	✓	11	✓	12	✓	31
P2	✗	41	✓	33	✓	34	✓	112
P3	✗	21	✗	33	✓	31	✓	88
P4	✗	57	✗	41	✗	54	✓	123

Information-Flow Policies. We created a small benchmark of simple programs that distinguish different synchronous information-flow policies. See Figure 12.¹⁵ We checked the following properties: **(OD)** is the observational determinism property stating that the output is identical among all paths, *i.e.*, $\forall \pi. \forall \pi'. \Box(o_\pi \leftrightarrow o_{\pi'})$. **(NI)** is a simple formulation of non-interference due to Goguen and Meseguer [GM82] that states that the output is fully determined by the low-security inputs, *i.e.*, $\forall \pi. \forall \pi'. \Box(l_\pi \leftrightarrow l_{\pi'}) \rightarrow \Box(o_\pi \leftrightarrow o_{\pi'})$. **(simSec)** is simulation-based security as discussed in Section 4.3. **(approxGNI₃)** is the approximation of generalized non-interference with fixed lookahead of 3 as discussed in Section 4.2. The results and running times for each instance (obtained using `hyperatlmc`) are given in Table 1.

Asynchronous Hyperproperties. Our model checker implements the transformation of a game structure to include an asynchronous scheduler (*cf.* Definition 5.1). Using `hyperatlmc`, we checked synchronous observational-determinism **(OD)** and asynchronous versions of observational-determinism **(OD_{asynch})** and non-interference **(NI_{asynch})**. Note that while **(OD_{asynch})** is expressible in the decidable fragment of AHLTL, **(NI_{asynch})** is not an admissible formula (and cannot be handled in [BCB⁺21]). As non-interference only requires the outputs to align provided the inputs do, one needs to take care that the asynchronous scheduler does

¹⁵We choose very simple programs to easily distinguish between the different security notions. Our tool `hyperatlmc` can handle more complex programs with larger bitwidths.

TABLE 2. Model checking results of synchronous and asynchronous properties (expressed in HyperATL^{*}) on small `while` programs. We give the model checking result (Res) (✓ indicates that the property holds, and ✗ that it is violated) and time taken by `hyperatlmc` in milliseconds (t). Program Q1 is the program from Figure 2, and Q2 is a slight modification that sets the output according to the low-security input but delays this update.

Program	(OD)		(OD _{asynch})		(NI _{asynch})	
	Res	t	Res	t	Res	t
Q1	✗	54	✓	97	✓	121
Q2	✗	61	✗	87	✓	95

not “cheat” by deliberately misaligning inputs and thereby invalidating the premise of this implication. Our results are given in Table 2.

9. RELATED WORK

The Landscape of Hyperproperties. There has been a lot of recent interest in logics for hyperproperties. Most logics are obtained by extending standard temporal or first-order/second-order logics with either path quantification or by a special equal-level predicate [FZ17]. See [CFHH19] for an overview. To the best of our knowledge, none of these logics can express strategic hyperproperties in multi-agent systems. In [FMSZ17, MSZ18], the authors study the verification of first-order HyperLTL on multi-agent workflow systems specified as first-order transition systems. The logic that is used (first-order HyperLTL) does not reason about the strategic behavior in the multi-agent systems.

Hyperproperties in Multi-Agent Systems. The approach taken in HyperATL^{*} of resolving the paths that are quantified in the prefix *incrementally* is only one possible angle to express hyperproperties in multi-agent systems. One could also envision a logic, that can state the existence of a strategy *with respect to* a hyperproperty, *i.e.*, state the existence of a strategy such that the set of plays compatible with this strategy satisfies a hyperproperty. Model checking of the resulting logic would subsume HyperLTL realizability, which is known to be undecidable even for simple alternation-free formulas [FHL⁺18]. The incremental approach in HyperATL^{*} is restrictive enough to maintain decidable model checking and powerful enough to express many properties of interest and subsume many existing logics (see Figure 1).

Epistemic Logics. The relationship between epistemic logics and hyperlogics is interesting, as both reason about the flow of information in a system. HyperLTL and LTL_K (LTL extended with a knowledge operator [FHMV95]) have incomparable expressiveness [BMP15]. In HyperQPTL [Rab16, BF23b] – which extends HyperLTL with additional propositional quantification – the knowledge operator can be encoded by explicitly marking the knowledge positions via propositional quantification [Rab16, §7]. By allowing second-order quantification, one can even reason about *common* knowledge in a system [HM90, BFFM23]. Alternating-time temporal logic has also been extended with knowledge operators [vdHW03].

The resulting logic, **ATEL**, can express properties of the form “if ξ knows ϕ , then she can enforce ψ via a strategy.” The natural extension of **ATEL** that allows for arbitrary nesting of quantification and temporal operators (*i.e.*, an extension of **ATL*** instead of **ATL**), is incomparable to **HyperATL***.

Model Checking. Decidable model checking is a crucial prerequisite for the effective use of a logic. Many of the existing (synchronous) hyperlogics admit decidable finite-state model checking, although mostly with non-elementary complexity [CFK⁺14]. Most hyperproperties encountered in practice can be expressed with few (if any) quantifier alternations. Alternation-free **HyperLTL** properties can be checked very efficiently by constructing the self-composition [BDR11], as, *e.g.*, implemented in the **MCHyper** tool [FRS15]. Properties with quantifier alternations can be checked by using automata complementations or language inclusion checks, as, *e.g.*, implemented in the **AutoHyper** tool [BF23a]. For properties in the $\forall^*\exists^*$ fragment, efficient approximations, such as the game-based approach [CFST19, BF22a], are applicable, even in infinite-state systems [BF22b]. For alternating-time temporal logic (in the non-hyper realm), model checking is efficient, especially when temporal operators cannot be nested as in **ATL** [AHK02, AHM⁺98]. In the presence of arbitrary nesting (as in **ATL***), model checking subsumes **LTL** realizability [PR89]. This causes a jump in the model checking complexity to 2-EXPTIME-completeness [AHK02]. **ATL** model checking has also been investigated in the presence of imperfect information [JÅ06, DT11, BGJ15, BMM17], and imperfect recall [Sch04]. Strategy logic [CHP10, MMPV14] (strictly) generalizes **ATL*** by considering strategies as first class objects that can be quantified. Model checking of strategy logic is decidable, but nonelementary-hard [CHP10, MMPV14].

Our lower bounds demonstrate that the combination of strategic quantification and hyperproperties results in a logic that is algorithmically harder (for model checking) than non-strategic hyperlogics (such as **HyperLTL**) or strategic (non-hyper) logics (such as **ATL***). The fragment of **HyperATL*** supported by **hyperatlmc** is algorithmically cheaper than full **HyperATL***; it is 2-EXPTIME-complete in the size of the specification and PTIME-complete in the size of the system.

Satisfiability. The satisfiability of a formula (*i.e.*, checking if a formula has a satisfying model) is relevant during the development of a specification. It can be used as a sanity check (to ensure that the specification is not already contradictory) or to determine implications between different specifications. The hardness of **HyperLTL** satisfiability can be characterized in terms of the structure of the quantifier prefix. Satisfiability is decidable (and EXPSpace-complete) for formulas in the $\exists^*\forall^*$ fragment and undecidable for all prefixes that contain a $\forall\exists$ alternation [FH16]. For **HyperCTL***, alternation-free formulas (where the quantifier structure is defined with respect to the scope of quantifiers in a negation-normal form) are decidable [Hah21]. However, already $\exists^*\forall^*$ formulas lead to undecidability as quantification can occur at all points along a path (by placing quantification below a \Box) and create a comb-like structure that can “simulate” a $\forall\exists$ alternation [Hah21]. Fortin et al. show that satisfiability of **HyperLTL** and **HyperCTL*** is highly undecidable; deciding satisfiability of a **HyperLTL** formula is Σ_1^1 -complete and deciding satisfiability of a **HyperCTL*** formula is Σ_1^2 -complete [FKTZ21]. By restricting the body of a formula and distinguishing between hyperproperty and trace property, one can identify classes of **HyperLTL** within the $\forall^*\exists^*$ fragment for which satisfiability remains decidable [BCF⁺22]. **ATL*** satisfiability was studied

by Schewe [Sch08] and found to coincide with the model-checking complexity (2-EXPTIME-complete in the size of the specification). This is surprising as for most branching-time temporal logics (such as CTL and CTL^{*}), satisfiability is strictly (at least exponentially) harder than model checking (in the size of the specification). As HyperATL^{*} subsumes HyperCTL^{*}, it inherits the Σ_1^2 -hardness of HyperCTL^{*} satisfiability.¹⁶ Identifying fragments of HyperATL^{*} that are decidable (or sit below the general Σ_1^2 -hardness) is interesting future work.

Asynchronous Hyperproperties. Extending hyperlogics to express asynchronous properties has only recently started to gain momentum [GMO21, BCB⁺21, BPS21, BF22b]. Baumeister et al. introduce AHLTL by extending HyperLTL with explicit trajectory quantification [BCB⁺21]. Gutsfeld et al. introduced a variant of the polyadic μ -calculus, called H_μ , and accompanying asynchronous automata that are able to express asynchronous hyperproperties [GMO21]. Bozzelli et al. present HyperLTL_S by extending HyperLTL with new modalities that remove redundant (for example stuttering) parts of a trace [BPS21]. Finite-state model checking for the logics proposed in [GMO21, BCB⁺21, BPS21] is undecidable. Observation-based HyperLTL [BF22b], can be seen as fragment of HyperLTL_S that is geared towards automated verification and admits decidable finite-state model checking. We can obtain decidable fragments of H_μ [GMO21] and HyperLTL_S [BPS21] by bounding the asynchronous offset by a constant k , *i.e.*, asynchronous execution may not run apart (“diverge”) for more than k steps. The (known) decidable fragment of AHLTL [BCB⁺21] can be encoded into HyperATL^{*} (Section 5.4).

10. CONCLUSION

We have introduced HyperATL^{*}, a temporal logic to express hyperproperties in multi-agent systems. Besides the obvious benefits of simultaneously reasoning about strategic choice and information flow, HyperATL^{*} provides a natural formalism to express *asynchronous* hyperproperties. Despite the added expressiveness, HyperATL^{*} model checking remains decidable for the entire logic. Its expressiveness and decidability, as well as the availability of practical model checking algorithms, make it a very promising choice for model checking tools for hyperproperties.

REFERENCES

- [AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5), 2002. doi:10.1145/585265.585270.
- [AHM⁺98] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: modularity in model checking. In *International Conference on Computer Aided Verification, CAV 1998*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998. doi:10.1007/BFb0028774.
- [BCB⁺21] Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In *International Conference on Computer Aided Verification, CAV 2021*, volume 12759 of *Lecture Notes in Computer Science*. Springer, 2021. doi:10.1007/978-3-030-81685-8_33.

¹⁶As alternating-time logics are evaluated over game structures, the satisfiability problem can either be stated as the search for a set of agents (containing the agents referred to in the formula) *and* game structure over those agents or the search for a game structure given a fixed set of agents (as part of the input). See [WLWW06] for details in the context of ATL. We assume that the set of agents is provided with the input.

- [BCF⁺22] Raven Beutner, David Carral, Bernd Finkbeiner, Jana Hofmann, and Markus Krötzsch. Deciding hyperproperties combined with functional specifications. In *Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2022*. ACM, 2022. doi:10.1145/3531130.3533369.
- [BDR11] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. Secure information flow by self-composition. *Math. Struct. Comput. Sci.*, 21(6), 2011. doi:10.1017/S0960129511000193.
- [BF21] Raven Beutner and Bernd Finkbeiner. A temporal logic for strategic hyperproperties. In *International Conference on Concurrency Theory, CONCUR 2021*, volume 203 of *LIPICs*. Schloss Dagstuhl, 2021. doi:10.4230/LIPICs.CONCUR.2021.24.
- [BF22a] Raven Beutner and Bernd Finkbeiner. Prophecy variables for hyperproperty verification. In *IEEE Computer Security Foundations Symposium, CSF 2022*. IEEE, 2022. doi:10.1109/CSF54842.2022.9919658.
- [BF22b] Raven Beutner and Bernd Finkbeiner. Software verification of hyperproperties beyond k-safety. In *International Conference on Computer Aided Verification, CAV 2022*, volume 13371 of *Lecture Notes in Computer Science*. Springer, 2022. doi:10.1007/978-3-031-13185-1_17.
- [BF23a] Raven Beutner and Bernd Finkbeiner. AutoHyper: Explicit-state model checking for HyperLTL. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023*, volume 13993 of *Lecture Notes in Computer Science*. Springer, 2023. doi:10.1007/978-3-031-30823-9_8.
- [BF23b] Raven Beutner and Bernd Finkbeiner. Model checking omega-regular hyperproperties with AutoHyperQ. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2023*, EPiC Series in Computing. EasyChair, 2023.
- [BFFM23] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger. Second-order hyperproperties. In *International Conference on Computer Aided Verification, CAV 2023*, Lecture Notes in Computer Science. Springer, 2023.
- [BGJ15] Nils Bulling, Valentin Goranko, and Wojciech Jamroga. Logics for reasoning about strategic abilities in multi-player games. In *Models of Strategic Reasoning - Logics, Games, and Communities*, volume 8972 of *Lecture Notes in Computer Science*. Springer, 2015. doi:10.1007/978-3-662-48540-8_4.
- [BKR10] Udi Boker, Orna Kupferman, and Adin Rosenberg. Alternation removal in büchi automata. In *International Colloquium on Automata, Languages and Programming, ICALP 2010*, volume 6199 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-14162-1_7.
- [BMM17] Raphaël Berthon, Bastien Maubert, and Aniello Murano. Decidability results for ATL* with imperfect information and perfect recall. In *Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017*. ACM, 2017.
- [BMP15] Laura Bozzelli, Bastien Maubert, and Sophie Pinchinat. Unifying hyper and epistemic temporal logics. In *International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2015*, volume 9034 of *Lecture Notes in Computer Science*. Springer, 2015. doi:10.1007/978-3-662-46678-0_11.
- [BPS21] Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of HyperLTL. In *Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*. IEEE, 2021. doi:10.1109/LICS52264.2021.9470583.
- [CFHH19] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. The hierarchy of hyperlogics. In *Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019*. IEEE, 2019. doi:10.1109/LICS.2019.8785713.
- [CFK⁺14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust, POST 2014*, volume 8414 of *Lecture Notes in Computer Science*. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.
- [CFST19] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In *International Conference on Computer Aided Verification, CAV 2019*, volume 11561 of *Lecture Notes in Computer Science*. Springer, 2019. doi:10.1007/978-3-030-25540-4_7.
- [CHP10] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. Strategy logic. *Inf. Comput.*, 208(6), 2010. doi:10.1016/j.ic.2009.07.004.
- [CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6), 2010. doi:10.3233/JCS-2009-0393.

- [DH94] Doron Drusinsky and David Harel. On the power of bounded concurrency I: finite automata. *J. ACM*, 41(3), 1994. doi:10.1145/176584.176587.
- [DT11] Catalin Dima and Ferucio Laurentiu Tiplea. Model-checking ATL under imperfect information and perfect recall semantics is undecidable. *CoRR*, abs/1102.4225, 2011. arXiv:1102.4225.
- [EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1), 1986. doi:10.1145/4904.4999.
- [FH16] Bernd Finkbeiner and Christopher Hahn. Deciding hyperproperties. In *International Conference on Concurrency Theory, CONCUR 2016*, volume 59 of *LIPIcs*. Schloss Dagstuhl, 2016. doi:10.4230/LIPIcs.CONCUR.2016.13.
- [FHL⁺18] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. Synthesizing reactive systems from hyperproperties. In *International Conference on Computer Aided Verification, CAV 2018*, volume 10981 of *Lecture Notes in Computer Science*. Springer, 2018. doi:10.1007/978-3-319-96145-3_16.
- [FHMV95] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995. doi:10.7551/mitpress/5803.001.0001.
- [FKTZ21] Marie Fortin, Louwe B. Kuijter, Patrick Totzke, and Martin Zimmermann. HyperLTL satisfiability is Σ_1^1 -complete, HyperCTL* satisfiability is Σ_1^2 -complete. In *International Symposium on Mathematical Foundations of Computer Science, MFCS 2021*, volume 202 of *LIPIcs*. Schloss Dagstuhl, 2021. doi:10.4230/LIPIcs.MFCS.2021.47.
- [FL09] Oliver Friedmann and Martin Lange. Solving parity games in practice. In *International Symposium on Automated Technology for Verification and Analysis, ATVA 2009*, volume 5799 of *Lecture Notes in Computer Science*. Springer, 2009. doi:10.1007/978-3-642-04761-9_15.
- [FMSZ17] Bernd Finkbeiner, Christian Müller, Helmut Seidl, and Eugen Zalinescu. Verifying security policies in multi-agent workflows with loops. In *ACM Conference on Computer and Communications Security, CCS 2017*. ACM, 2017. doi:10.1145/3133956.3134080.
- [FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *International Conference on Computer Aided Verification, CAV 2015*, volume 9206 of *Lecture Notes in Computer Science*. Springer, 2015. doi:10.1007/978-3-319-21690-4_3.
- [FZ17] Bernd Finkbeiner and Martin Zimmermann. The first-order logic of hyperproperties. In *Symposium on Theoretical Aspects of Computer Science, STACS 2017*, volume 66 of *LIPIcs*. Schloss Dagstuhl, 2017. doi:10.4230/LIPIcs.STACS.2017.30.
- [GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy, SP 1982*. IEEE, 1982. doi:10.1109/SP.1982.10014.
- [GMO21] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL), 2021. doi:10.1145/3434319.
- [Hah21] Christopher Hahn. *Logical and deep learning methods for temporal reasoning*. PhD thesis, Saarland University, 2021. doi:10.22028/D291-35192.
- [HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3), 1990. doi:10.1145/79147.79161.
- [HWS06] Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *IEEE Computer Security Foundations Workshop, CSFW 2006*. IEEE, 2006. doi:10.1109/CSFW.2006.6.
- [JÅ06] Wojciech Jamroga and Thomas Ågotnes. What agents can achieve under incomplete information. In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)*. ACM, 2006. doi:10.1145/1160633.1160672.
- [KMSZ18] Jan Kretínský, Tobias Meggendorfer, Salomon Sickert, and Christopher Ziegler. Rabinizer 4: From LTL to your favourite deterministic automaton. In *International Conference on Computer Aided Verification, CAV 2018*, volume 10981 of *Lecture Notes in Computer Science*. Springer, 2018. doi:10.1007/978-3-319-96145-3_30.
- [McC88] Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy, SP 1988*. IEEE, 1988. doi:10.1109/SECPRI.1988.8110.
- [MH84] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32, 1984. doi:10.1016/0304-3975(84)90049-5.

- [Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.
- [MMPV14] Fabio Mogavero, Aniello Murano, Giuseppe Perelli, and Moshe Y. Vardi. Reasoning about strategies: On the model-checking problem. *ACM Trans. Comput. Log.*, 15(4), 2014. doi:10.1145/2631917.
- [MS10] Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *European Symposium on Research in Computer Security, ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-15497-3_8.
- [MSS88] David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Annual Symposium on Logic in Computer Science, LICS 1988*. IEEE, 1988. doi:10.1109/LICS.1988.5139.
- [MSZ18] Christian Müller, Helmut Seidl, and Eugen Zalinescu. Inductive invariants for noninterference in multi-agent workflows. In *IEEE Computer Security Foundations Symposium, CSF 2018*. IEEE Computer Society, 2018. doi:10.1109/CSF.2018.00025.
- [PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Annual ACM Symposium on Principles of Programming Languages, POPL 1989*. ACM Press, 1989. doi:10.1145/75277.75293.
- [Rab16] Markus N. Rabe. *A temporal logic approach to information-flow control*. PhD thesis, Saarland University, 2016.
- [Sab03] Andrei Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *International Andrei Ershov Memorial Conference, PSI 2003*, volume 2890 of *Lecture Notes in Computer Science*. Springer, 2003. doi:10.1007/978-3-540-39866-0_27.
- [Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2), 1970. doi:10.1016/S0022-0000(70)80006-X.
- [Sch04] Pierre-Yves Schobbens. Alternating-time logic with imperfect recall. *Electron. Notes Theor. Comput. Sci.*, 85(2), 2004. doi:10.1016/S1571-0661(05)82604-0.
- [Sch08] Sven Schewe. ATL* satisfiability is 2EXPTIME-complete. In *International Colloquium on Automata, Languages and Programming, ICALP 2008*, volume 5126 of *Lecture Notes in Computer Science*. Springer, 2008. doi:10.1007/978-3-540-70583-3_31.
- [SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *IEEE Computer Security Foundations Workshop, CSFW 2000*. IEEE, 2000. doi:10.1109/CSFW.2000.856937.
- [Sti95] Colin Stirling. Modal and temporal logics for processes. In *Banff Higher Order Workshop on Logics for Concurrency - Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*. Springer, 1995. doi:10.1007/3-540-60915-6_5.
- [Sto74] Larry Joseph Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Massachusetts Institute of Technology, 1974.
- [vdHW03] Wiebe van der Hoek and Michael J. Wooldridge. Cooperation, knowledge, and time: Alternating-time temporal epistemic logic and its applications. *Stud Logica*, 75(1), 2003. doi:10.1023/A:1026185103185.
- [VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1), 1994. doi:10.1006/inco.1994.1092.
- [WJ90] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *IEEE Symposium on Security and Privacy, SP 1990*. IEEE, 1990. doi:10.1109/RISP.1990.63846.
- [WLWW06] Dirk Walther, Carsten Lutz, Frank Wolter, and Michael J. Wooldridge. ATL satisfiability is indeed EXPTIME-complete. *J. Log. Comput.*, 16(6), 2006. doi:10.1093/logcom/ex1009.

APPENDIX A.

In this section, we show Proposition 6.1 for the most interesting case where $\varphi = \langle\langle A \rangle\rangle \pi. \psi$. For the correctness of the construction in the other cases see *e.g.*, [MSS88, FRS15]. For simplicity, we assume that \mathcal{G} is a CGS. Let \mathcal{A}_ψ be the inductively constructed automaton for ψ that, by the induction hypothesis, is \mathcal{G} -equivalent to ψ . We consider the construction

in Figure 5. We show that \mathcal{A}_φ is \mathcal{G} -equivalent to φ by showing both directions of the “iff” in the definition of \mathcal{G} -equivalence separately.

Lemma A.1. *For any $p_1, \dots, p_n \in S^\omega$, if $\otimes(p_1, \dots, p_n) \in \mathcal{L}(\mathcal{A}_\varphi)$ then $[\pi_i \mapsto p_i]_{i=1}^n \models_{\mathcal{G}} \varphi$*

Proof. Let (T, r) be an accepting run of \mathcal{A}_φ on $\otimes(p_1, \dots, p_n)$. We use the disjunctive choices made in (T, r) to construct strategies $F_A = \{f_\xi \mid \xi \in A\}$ where $f_\xi : S^+ \rightarrow \mathcal{M}$. For each element $u \in S^+$ we define $f_\xi(u)$ for each $\xi \in A$ as follows (it is important to construct the response to u together as the path identified next is not be unique). Let $u = u(0) \dots u(k)$. We check if there exists a node τ in (T, r) such that the nodes along τ are labeled by u , *i.e.*,

$$r(\epsilon), r(\tau[0, 0]), r(\tau[0, 1]), \dots, r(\tau[0, |\tau| - 1]) = q_{init}, (u(1), -), \dots, (u(k), -).$$

Note that the offset is intentional, *i.e.*, the first element $u(0)$ does not occur in τ (as \mathcal{A}_φ skips over s_n° in the first step). If no such node exists, we define $f_\xi(u)$ arbitrarily for all $\xi \in A$ (any play that is compatible with the strategy never reaches this situation). Otherwise, let $r(\tau) = (u(k), q)$ where q is a state of \mathcal{A}_ψ^{det} (or $r(\tau) = q_{init}$ if $|u| = 1$). By construction of $\tilde{\mathcal{A}}_\varphi$ we have that the children of τ satisfy the formula

$$\bigvee_{\sigma : A \rightarrow \mathcal{M}} \bigwedge_{\sigma' : \bar{A} \rightarrow \mathcal{M}} (\delta(u(k), \sigma + \sigma'), \rho(q, [p_1(k), \dots, p_n(k), u(k)])).$$

(The case where $r(\tau) = q_{init}$ is analogous.) There must thus exist (at least one) $\sigma_u : A \rightarrow \mathcal{M}$ such that for every $\sigma' : \bar{A} \rightarrow \mathcal{M}$ there is a child of τ labeled with

$$(\delta(u(k), \sigma_u + \sigma'), \rho(q, [p_1(k), \dots, p_n(k), u(k)])).$$

We define

$$f_\xi(u) := \sigma_u(\xi)$$

for each $\xi \in A$. By assumption of σ_u for any $\sigma' : \bar{A} \rightarrow \mathcal{M}$, there is a successor of τ labeled by $\delta(u(k), \sigma_u + \sigma')$, *i.e.*, all adversarial moves lead to a node in (T, r) if agents in A play σ_u .

It is, therefore, easy to see that for all $p \in out(\mathcal{G}, s_n^\circ, F_A)$ (where $s_n^\circ := s_0$ if $n = 0$ and $s_n^\circ := p_n(0)$ otherwise), there exist a path in (T, r) labeled with $q_{init}(p(1), q_1)(p(2), q_2) \dots$. By definition of $\tilde{\rho}$, the sequence of automaton state q_0, q_1, q_2, \dots (where q_0 is the initial state of \mathcal{A}_ψ^{det}) is the unique run of \mathcal{A}_ψ^{det} on $\otimes(p_1, \dots, p_n, p)$. As (T, r) is accepting this sequence of automata states is accepting, we thus get that $\otimes(p_1, \dots, p_n, p) \in \mathcal{L}(\mathcal{A}_\psi^{det}) = \mathcal{L}(\mathcal{A}_\psi)$. By the induction hypothesis (from the proof of Proposition 6.1) we have that \mathcal{A}_ψ is \mathcal{G} -equivalent to ψ and so $[\pi_i \mapsto p_i]_{i=1}^n \cup [\pi \mapsto p] \models_{\mathcal{G}} \psi$. As this holds for all $p \in out(\mathcal{G}, s_n^\circ, F_A)$, F_A is a winning set of strategies and $[\pi_i \mapsto p_i]_{i=1}^n \models_{\mathcal{G}} \varphi$ by the semantics of HyperATL*. \square

Lemma A.2. *For any $p_1, \dots, p_n \in S^\omega$, if $[\pi_i \mapsto p_i]_{i=1}^n \models_{\mathcal{G}} \varphi$ then $\otimes(p_1, \dots, p_n) \in \mathcal{L}(\mathcal{A}_\varphi)$*

Proof. Let $F_A = \{f_\xi \mid \xi \in A\}$ be a winning strategy for the agents in A , *i.e.*, for all $p \in out(\mathcal{G}, s_n^\circ, F_A)$, $[\pi_i \mapsto p_i]_{i=1}^n \cup [\pi \mapsto p] \models_{\mathcal{G}} \psi$. We construct an accepting run (T, r) of \mathcal{A}_φ on $\otimes(p_1, \dots, p_n)$. We construct this tree incrementally by adding children to existing nodes.

The root ϵ is labeled by q_{init} . Now let $\tau \in T$ be any node in the tree constructed so far and let

$$r(\epsilon), r(\tau[0, 0]), r(\tau[0, 1]), \dots, r(\tau[0, |\tau| - 1]) = q_{init}, (s_1, q_1), \dots, (s_k, q_k)$$

be the label of the nodes along τ . We define the move vector $\sigma_\tau : A \rightarrow \mathcal{M}$ via $\sigma_\tau(\xi) := f_\xi(s_n^\circ, s_1, \dots, s_k)$ for each $\xi \in A$ (where $s_n^\circ := s_0$ if $n = 0$ and $s_n^\circ := p_n(0)$ otherwise). For each move vectors $\sigma' : \bar{A} \rightarrow \mathcal{M}$ we add a new child of τ labeled with

$$(\delta(s_k, \sigma_\tau + \sigma'), \rho(q, [p_1(|\tau|), \dots, p_n(|\tau|), s_k])).$$

By construction of the transition function of \mathcal{A}_φ , those children satisfy the transition relation ρ' (see Figure 5).

The constructed tree (T, r) is thus a run on $\otimes(p_1, \dots, p_n)$. We now claim that (T, r) is accepting. Consider any infinite path in (T, r) labeled by $q_{init}(s_1, q_1)(s_2, q_2), \dots$. By construction of the tree, it is easy to see that the path $p = s_n^\circ, s_1, s_2, \dots$ is contained in $out(\mathcal{G}, s_n^\circ, F_A)$ as all children added to a node were added in accordance with F_A . As F_A is winning and by the **HyperATL*** semantics, we get that $[\pi_i \mapsto p_i]_{i=1}^n \cup [\pi \mapsto p] \models_{\mathcal{G}} \psi$. By induction hypothesis (from the proof of Proposition 6.1) we get that \mathcal{A}_ψ is \mathcal{G} -equivalent to ψ so $\otimes(p_1, \dots, p_n, p) \in \mathcal{L}(\mathcal{A}_\psi) = \mathcal{L}(\mathcal{A}_\psi^{det})$. By construction of \mathcal{A}_φ the automaton sequence q_0, q_1, q_2, \dots (where q_0 is the initial state of \mathcal{A}_ψ^{det}) is the unique run of \mathcal{A}_ψ^{det} on $\otimes(p_1, \dots, p_n, p)$ and therefore accepting. So (T, r) is accepting, and it follows that $\otimes(p_1, \dots, p_n) \in \mathcal{L}(\mathcal{A}_\varphi)$. \square