Runtime Enforcement of Hyperproperties*

Norine Coenen[1], Bernd Finkbeiner[1], Christopher Hahn[1], Jana Hofmann[1], and Yannick Schillo[2]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany,
{norine.coenen, finkbeiner, christopher.hahn, jana.hofmann}@cispa.de
[2] Saarland University, Saarbrücken, Germany,
s8yaschi@stud.uni-saarland.de

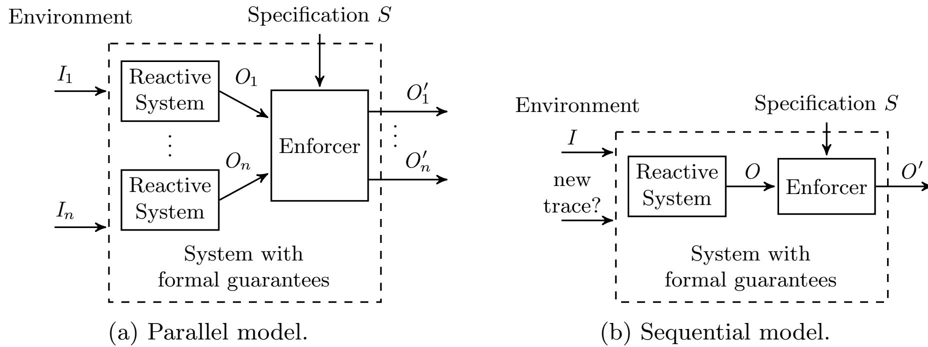
Abstract. An enforcement mechanism monitors a reactive system for undesired behavior at runtime and corrects the system's output in case it violates the given specification. In this paper, we study the enforcement problem for *hyperproperties*, i.e., properties that relate multiple computation traces to each other. We elaborate the notion of *sound* and *transparent* enforcement mechanisms for hyperproperties in two trace input models: 1) the parallel trace input model, where the number of traces is known a-priori and all traces are produced and processed in parallel and 2) the sequential trace input model, where traces are processed sequentially and no a-priori bound on the number of traces is known. For both models, we study enforcement algorithms for specifications given as formulas in universally quantified HyperLTL, a temporal logic for hyperproperties. For the parallel model, we describe an enforcement mechanism based on parity games. For the sequential model, we show that enforcement is in general undecidable and present algorithms for reasonable simplifications of the problem (partial guarantees or the restriction to safety properties). Furthermore, we report on experimental results of our prototype implementation for the parallel model.

1 Introduction

Runtime enforcement combines the strengths of dynamic and static verification by monitoring the output of a running system and also *correcting* it in case it violates a given specification. Enforcement mechanisms thus provide formal guarantees for settings in which a system needs to be kept alive while also fulfilling critical properties. Privacy policies, for example, cannot be ensured by shutting down the system to prevent a leakage: an attacker could gain information just from the fact that the execution stopped.

Runtime enforcement has been successfully applied in settings where specifications are given as *trace properties* [16,14]. Not every system behavior, however, can be specified as a trace property. Many security and privacy policies are

* 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 European Research Council (ERC) Grant OSARES (No. 683300).



(a) Parallel model. (b) Sequential model.

Fig. 1: Runtime enforcement for a reactive system. In case the input-output-relation would violate the specification S, the enforcer corrects the output.

hyperproperties [6], which generalize trace properties by relating multiple execution traces to each other. Examples are noninterference [37,30], observational determinism [41], and the detection of out-of-the-ordinary values in multiple data streams [27]. Previous work on runtime enforcement of hyperproperties either abstractly studied the class of enforceable hyperproperties [32] or security policies [38], or provided solutions for specific security policies like noninterference [38,12,28]. Our contribution is two-fold.

First, conceptually, we show that hyperproperty enforcement of reactive systems needs to solve challenging variants of the synthesis problem and that the concrete formulation depends on the given trace input model. We distinguish two input models 1) the parallel trace input model, where the number of traces is known a-priori and all traces are produced and processed in parallel and 2) the sequential trace input model, where traces are processed sequentially and no a-priori bound on the number of traces is known. Figure 1 depicts the general setting in these input models. In the parallel trace input model, the enforcement mechanism observes n traces at the same time. This is, for example, the natural model if a system runs in secure multi-execution [10]. In the sequential trace input model, system runs are observed in sessions, i.e., one at a time. An additional input indicates that a new session (i.e., trace) starts. Instances of this model naturally appear, for example, in web-based applications.

Second, algorithmically, we describe enforcement mechanisms for a concrete specification language. The best-studied temporal logic for hyperproperties is HyperLTL [7]. It extends LTL with trace variables and explicit trace quantification to relate multiple computation traces to each other. HyperLTL can express many standard information flow policies [7]. In particular, it is flexible enough to state different application-tailored specifications. We focus on universally quantified formulas, a fragment in which most of the enforceable hyperproperties naturally reside. For both trace input models, we develop enforcement mechanisms based on parity game solving. For the sequential model, we show that the problem is undecidable in general but provide algorithms for the simpler

case that the enforcer only guarantees a correct continuation for the rest of the current session. Furthermore, we describe an algorithm for the case that the specification describes a safety property. Our algorithms monitor for *losing* prefixes, i.e., so-far observed traces for which the system has no winning strategy against an adversarial environment. We ensure that our enforcement mechansims are *sound* by detecting losing prefixes at the earliest possible point in time. Furthermore, they are *transparent*, i.e., non-losing prefixes are not altered.

We accompany our findings with a prototype implementation for the parallel model and conduct two experiments: 1) we enforce symmetry in mutual exclusion algorithms and 2) we enforce the information flow policy observational determinism. We will see that enforcing such complex HyperLTL specifications can scale to large traces once the initial parity game solving succeeds.

Related Work. HyperLTL has been studied extensively, for example, its expressiveness [5,8] as well as its verification [7,22,9,21], synthesis [18], and monitoring problem [39,19,20,24,4]. Especially relevant is the work on realizability monitoring for LTL [11] using parity games. Existing work on runtime enforcement includes algorithms for safety properties [3,40], real-time properties [36,17], concurrent software specifications [29], and concrete security policies [38,12,28]. For a tutorial on variants of runtime enforcement see [14]. Close related work is [32], which also studies the enforcement of general hyperproperties but independently of a concrete specification language (in contrast to our work). Systems are also assumed to be reactive and black-box, but there is no distinction between different trace input models. While we employ parity game solving, their enforcement mechanism executes several copies of the system to obtain executions that are related by the specification.

2 Preliminaries

Let Σ be an alphabet of atomic propositions. We assume that Σ can be partitioned into inputs and outputs, i.e., $\Sigma = I \mathbin{\dot\cup} O$. A finite sequence $t \in (2^{\Sigma})^*$ is a *finite trace*, an infinite sequence $t \in (2^{\Sigma})^{\omega}$ is an *infinite trace*. We write $t[i]$ for the $(i+1)$-th position of a trace, $t[0..i]$ for its prefix of length $i+1$, and $t[i,\infty]$ for the suffix from position i. A hyperproperty H is a set of sets of infinite traces.

HyperLTL. HyperLTL [7] is a linear temporal hyperlogic that extends LTL [35] with prenex trace quantification. The syntax of HyperLTL is given with respect to an alphabet Σ and a set $\mathcal{V}$ of trace variables.

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

where $a \in \Sigma$ and $\pi \in \mathcal{V}$. The atomic proposition a is indexed with the trace variable π it refers to. We assume that formulas contain no free trace variables. HyperLTL formulas are evaluated on a set $T \subseteq (2^{\Sigma})^{\omega}$ of infinite traces and a

trace assignment function $\Pi : \mathcal{V} \to T$. We use $\Pi[\pi \mapsto t]$ for the assignment that returns $\Pi(\pi')$ for $\pi' \neq \pi$ and t otherwise. Furthermore, let $\Pi[i,\infty]$ be defined as $\Pi[i,\infty](\pi) = \Pi(\pi)[i,\infty]$. The semantics of HyperLTL is defined as follows:

$$
\begin{array}{ll}
T, \Pi \models a_\pi & \text{iff } a \in \Pi(\pi)[0] \\
T, \Pi \models \neg\psi & \text{iff } T, \Pi \not\models \psi \\
T, \Pi \models \psi_1 \vee \psi_2 & \text{iff } T, \Pi \models \psi_1 \text{ or } T, \Pi \models \psi_2 \\
T, \Pi \models \bigcirc \psi & \text{iff } T, \Pi[1,\infty] \models \psi \\
T, \Pi \models \psi_1 \mathcal{U} \psi_2 & \text{iff } \exists i \geq 0.\, T, \Pi[i,\infty] \models \psi_2 \text{ and } \forall 0 \leq j < i.\, T, \Pi[j,\infty] \models \psi_1 \\
T, \Pi \models \exists \pi.\, \varphi & \text{iff } \exists t \in T.\, T, \Pi[\pi \mapsto t] \models \varphi \\
T, \Pi \models \forall \pi.\, \varphi & \text{iff } \forall t \in T.\, T, \Pi[\pi \mapsto t] \models \varphi
\end{array}
$$

We also use the derived boolean connectives $\wedge, \rightarrow, \leftrightarrow$ as well as the derived temporal operators $\Diamond \varphi \equiv \mathit{true} \, \mathcal{U} \, \varphi$, $\Box \varphi \equiv \neg(\Diamond \neg \varphi)$, and $\varphi \, \mathcal{W} \, \psi \equiv (\varphi \, \mathcal{U} \, \psi) \vee \Box \varphi$. A trace set T satisfies a HyperLTL formula φ if $T, \emptyset \models \varphi$, where $\emptyset$ denotes the empty trace assignment.

Parity Games. A parity game $\mathcal{G}$ is a two-player game on a directed graph arena, where the states $V = V_0 \mathbin{\dot\cup} V_1$ are partitioned among the two players P_0 and P_1. States belonging to P_0 and P_1 are required to alternate along every path. States are labeled with a coloring function $c : V \to \mathbb{N}$. Player P_0 wins the game if they have a strategy to enforce that the highest color occurring infinitely often in a run starting in the initial state is even. The winning region of a parity game is the set of states from which player P_0 has a winning strategy. A given LTL formula φ can be translated to a parity game $\mathcal{G}_\varphi$ in doubly-exponential time [13]. Formula φ is realizable iff player P_0 wins the game $\mathcal{G}_\varphi$. Its winning strategy σ_0 induces the reactive strategy σ representing a system implementation that satisfies φ.

3 Hyperproperty Enforcement

In this section, we develop a formal definition of hyperproperty enforcement mechanisms for reactive systems modeled with the parallel and the sequential trace input model. To this end, we first formally describe reactive systems under the two trace input models by the *prefixes* they can produce. Next, we develop the two basic requirements on enforcement mechanisms, *soundness* and *transparency* [14,15], for our settings. Soundness is traditionally formulated as

> the enforced system should be correct w.r.t. the specification.

Transparency (also known as precision [32]) states that

> the behavior of the system is modified in a minimal way, i.e., the longest correct prefix should be preserved by the enforcement mechanism.

In the context of reactive systems, formal definitions for soundness and transparency need to be formulated in terms of strategies that describe how the

(a) Prefix in the parallel model. (b) Prefix in the sequential model.

Fig. 2: Visualization of prefixes in trace input models.

enforcement mechanism reacts to the inputs from the environment and outputs produced by the system. We therefore define soundness and transparency based on the notion of *losing prefixes* (i.e. prefixes for that no winning strategy exists) inspired by work on monitoring reactive systems [11]. We will see that the definition of losing prefixes depends heavily on the chosen trace input model. Especially the sequential model defines an interesting new kind of synthesis problem, which varies significantly from the known HyperLTL synthesis problem.

As is common in the study of runtime techniques for reactive systems, we make the following reasonable assumptions. First, reactive systems are treated as *black boxes*, i.e., two reactive systems with the same observable input-output behavior are considered to be equal. Thus, enforcement mechanisms cannot base their decisions on implementation details. Second, w.l.o.g. and to simplify presentation, we assume execution traces to have *infinite length*. Finite traces can always be interpreted as infinite traces, e.g., by adding end^ω. To reason about finite traces, on the other hand, definitions like the semantics of HyperLTL would need to accommodate many special cases like traces of different lengths. Lastly, we assume that control stays with the enforcer after a violation occurred instead of only correcting the error and handing control back to the system afterwards. Since we aim to provide formal guarantees, these two problems are equivalent: if only the error was corrected, the enforcement mechanism would still need to ensure that the correction does not make the specification unrealizable in the future, i.e., it would need to provide a strategy how to react to all future inputs.

3.1 Trace Input Models

We distinguish two *trace input models* [19], the parallel and the sequential model. The trace input models describe how a reactive system is employed and how its traces are obtained (see Figure 1). We formally define the input models by the prefixes they can produce. The definitions are visualized in Figure 2. In the parallel model, a fixed number of n systems are executed in parallel, producing n events at a time.

Definition 1 (Prefix in the Parallel Model). *An n-tuple of finite traces $U = (u_1, \dots u_n) \in ((2^\Sigma)^*)^n$ is a prefix of $V = (v_1, \dots v_n) \in ((2^\Sigma)^{*/\omega})^n$ (written $U \preceq V$) in the parallel model with n traces iff each u_i is a prefix of v_i (also denoted by $u_i \preceq v_i$).*

The prefix definition models the allowed executions of a system under the parallel trace input model: If the system produces U and after a few more steps produces

V, then $U \preceq V$. Note that the prefix definition is transitive: U can be a prefix of another prefix (then the traces in V are of finite length) or a prefix of infinite-length traces.

In the sequential model, the traces are produced one by one and there is no a-priori known bound on the number of traces.

Definition 2 (Prefix in the Sequential Model). *Let $U = (u_1, \ldots, u_n) \in ((2^{\Sigma})^{\omega})^*$ be a sequence of traces and $u \in (2^{\Sigma})^*$ be a finite trace. Let furthermore $V = (v_1, \ldots, v_n, \ldots)$ be a (possibly infinite) sequence of traces with $v_i \in (2^{\Sigma})^{\omega}$, and $v \in (2^{\Sigma})^*$ be a finite trace. We call (U, u) a prefix of (V, v) (written $(U, u) \preceq (V, v)$) iff either 1) $U = V$ and $u \preceq v$ or 2) $V = u_1, \ldots, u_n, v_{n+1}, \ldots$ and $u \preceq v_{n+1}$.*

We additionally say that $(U, u) \preceq V$ if $(U, u) \preceq (V, \epsilon)$, where ϵ is the empty trace. To continue an existing prefix (U, u), the system either extends the started trace u or finishes u and continues with additional traces. Traces in U are of infinite length and describe finished sessions. This means that they cannot be modified after the start of a new session. Again, prefixes in this model are transitive and are also defined for infinite sets.

Remark 1. We defined prefixes tailored to the trace input models to precisely capture the influence of the models on the enforcement problem. Usually, a set of traces T is defined as prefix of a set of traces T' if and only if $\forall t \in T.\ \exists t' \in T'.\ t \preceq t'$ [6]. A prefix in the sequential model, however, *cannot* be captured by the traditional prefix definition, as it does not admit infinite traces in a prefix.

3.2 Losing Prefixes for Hyperproperties

Losing prefixes describe *when* an enforcer has to intervene based on possible strategies for future inputs. As we will see, the definition of losing prefixes, and thus the definition of the enforcement problem, differs significantly for both input models. For the rest of this section, let H denote an arbitrary hyperproperty.

We first define strategies for the parallel model with n parallel sessions. In the enforcement setting, a strategy receives a previously recorded prefix. Depending on that prefix, the enforcer's strategy might react differently to future inputs. We therefore define a *prefixed strategy* as a higher-order function $\sigma : ((2^{\Sigma})^*)^n \rightarrow ((2^I)^*)^n \rightarrow (2^O)^n$ over $\Sigma = I \mathbin{\dot{\cup}} O$. The strategy first receives a prefix (produced by the system), then a sequence of inputs on all n traces, and reacts with an output for all traces. We define a losing prefix as follows.

Definition 3 (Losing Prefix in the Parallel Model). *A strategy $\sigma(U)$ is losing for H with $U = (u_1, \ldots, u_n) \in ((2^{\Sigma})^*)^n$ if there are input sequences $(v_1, \ldots, v_n) \in ((2^I)^{\omega})^n$ such that the following set is not in H:*

$$\bigcup_{1 \leq i \leq n} \{u_i \cdot (v_i[0] \cup \sigma_U(\epsilon)(i)) \cdot (v_i[1] \cup \sigma_U(v_i[0])(i)) \cdot (v_i[2] \cup \sigma_U(v_i[0]v_i[1])(i)) \ldots\},$$

where $\sigma_U = \sigma(U)$ and $\sigma_U(\cdot)(i)$ denotes the i-th output that σ produces.

We say that $\sigma(U)$ is winning if it is not losing. A prefix U is winning if there is a strategy σ such that $\sigma(U)$ is winning. Lastly, σ is winning if $\sigma(\epsilon)$ is winning and for all non-empty winning prefixes U, $\sigma(U)$ is winning.

Similar to the parallel model, a prefixed strategy in the sequential model is a function $\sigma : ((2^{\Sigma})^{\omega})^{*} \times (2^{\Sigma})^{*} \to (2^{I})^{*} \to 2^{O}$ over $\Sigma = I \dot{\cup} O$. The definition of a losing prefix is the following.

Definition 4 (Losing Prefix in the Sequential Model). *In the sequential model, a strategy σ is losing with a prefix (U, u) for H, if there are input sequences $V = (v_0, v_1, \ldots)$ with $v_i \in (2^I)^{\omega}$, such that the set $U \cup \{t_0, t_1, \ldots\}$ is not in H, where $t_0, t_1, \ldots$ are defined as follows.*

$$\begin{aligned} t_0 &\coloneqq u \cdot (v_0[0] \cup \sigma(U, u)(\epsilon)) \cdot (v_0[1] \cup \sigma(U, u)(v_0[0])) \cdot \ldots \\ t_1 &\coloneqq (v_1[0] \cup \sigma(U \cup \{t_0\}, \epsilon)(\epsilon)) \cdot (v_1[1] \cup \sigma(U \cup \{t_0\}, \epsilon)(v_1[0])) \cdot \ldots \\ t_2 &\coloneqq (v_2[0] \cup \sigma(U \cup \{t_0, t_1\}, \epsilon)(\epsilon)) \cdot (v_2[1] \cup \sigma(U \cup \{t_0, t_1\}, \epsilon)(v_2[0])) \cdot \ldots \end{aligned}$$

Winning prefixes and strategies are defined analogously to the parallel model.

Remark 2. The above definitions illustrate that enforcing hyperproperties in the sequential model defines an intriguing but complex problem. Strategies react to inputs based on the observed prefix. The *same* input sequence can therefore be answered differently in the first session and, say, in the third session. The enforcement problem thus not simply combines monitoring and synthesis but formulates a different kind of problem.

3.3 Enforcement Mechanisms

With the definitions of the previous sections, we adapt the notions of sound and transparent enforcement mechanisms to hyperproperties under the two trace input models. We define an enforcement mechanism *enf* for a hyperproperty H to be a computable function which transforms a black-box reactive system S with trace input model $\mathcal{M}$ into a reactive system $\mathit{enf}(S)$ with the same input model.

Definition 5 (Soundness). *enf is* sound *if for all reactive systems S and all input sequences in model $\mathcal{M}$, the set of traces produced by $\mathit{enf}(S)$ is in H.*

Definition 6 (Transparency). *enf is* transparent *if the following holds: Let U be a prefix producible by S with input sequence s_I. If U is winning, then for any prefix V producible by $\mathit{enf}(S)$ with input sequence s'_I where $s_I \preceq s'_I$, it holds that $U \preceq V$.*

We now have everything in place to define when a hyperproperty is enforcable for a given input model.

Definition 7 (Enforceable Hyperproperties). *A hyperproperty H is* enforceable *if there is a sound and transparent enforcement mechanism.*

It is now straightforward to see that in order to obtain a sound and transparent enforcement mechanism, we need to construct a winning strategy for H.

Proposition 1. *Let H be a hyperproperty and $\mathcal{M}$ be an input model. Assume that it is decidable whether a prefix U is losing in model $\mathcal{M}$ for H. Then there exists a sound and transparent enforcement mechanism enf for H iff there exists a winning strategy in $\mathcal{M}$ for H.*

The above proposition describes how to construct enforcement algorithms: We need to solve the synthesis problem posed by the respective trace input model. However, we have to restrict ourselves to properties that can be monitored for losing prefixes. This is only natural: for example, the property expressed by the HyperLTL formula $\exists \pi. \Box a_\pi$ can in general not be enforced since it contains a hyperliveness [6] aspect: There is always the possibility for the required trace π to occur in a future session (c.f. monitorable hyperproperties in [1,19]). We therefore describe algorithms for HyperLTL specifications from the universal fragment $\forall \pi_1. \ldots \forall \pi_k. \varphi$ of HyperLTL. Additionally, we assume that the specification describes a property whose counterexamples have losing prefixes.

Before jumping to concrete algorithms, we describe two example scenarios of hyperproperty enforcement with different trace input models.

Example 1 (Fairness in Contract Signing). Contract signing protocols let multiple parties negotiate a contract. In this setting, fairness requires that in every situation where Bob can obtain Alice's signature, Alice must also be able to obtain Bob's signature. Due to the asymmetric nature of contract signing protocols (one party has to commit first), fairness is difficult to achieve (see, e.g., [33]). Many protocols rely on a trusted third party (TTP) to guarantee fairness. The TTP may negotiate multiple contracts in parallel sessions. The natural trace input model is therefore the parallel model. Fairness forbids the existence of two traces π and π' that have the same prefix of inputs, followed in π by Bob requesting (R^B) and receiving the signed contract (S^B), and in π' by Alice requesting (R^A), but *not* receiving the signed contract ($\neg S^A$):

$$\forall \pi. \forall \pi'. \neg((\bigwedge_{i \in I}(i_\pi \leftrightarrow i_{\pi'}))\ \mathcal{U}\ (R^B_\pi \wedge R^A_{\pi'} \wedge \bigcirc(S^B_\pi \wedge \neg S^A_{\pi'})))$$

Example 2 (Privacy in Fitness Trackers). Wearables track a wide range of extremely private health data which can leak an astonishing amount of insight into your health. For instance, it has been found that observing out-of-the-ordinary heart rate values correlates with diseases like the common cold or even Lyme disease [27]. Consider the following setting. A fitness tracker continuously collects data that is stored locally on the user's device. Additionally, the data is synced with an external cloud. While locally stored data should be left untouched, uploaded data has to be enforced to comply with information flow policies. Each day, a new stream of data is uploaded, hence the sequential trace input model would be appropriate. Comparing newer streams with older streams allows for the detection of anomalies. We formalize an exemplary property of this scenario

in HyperLTL. Let HR be the set of possible heart rates. Let furthermore *active* denote whether the user is currently exercising. Then the following property ensures that unusually high heart rate values are not reported to the cloud:

$$\forall \pi. \forall \pi'. \square(active_\pi \leftrightarrow active_{\pi'} \rightarrow \bigwedge_{r \in HR} (r_\pi \leftrightarrow r_{\pi'}))$$

4 Enforcement Algorithms for HyperLTL Specifications

For both trace input models, we present sound and transparent enforcement algorithms for universal HyperLTL formulas defining hyperproperties with losing prefixes. First, we construct an algorithm for the parallel input model based on parity game solving. For the sequential trace input model, we first show that the problem is undecidable in the general case. Next, we provide an algorithm that only finishes the remainder of the current session. This simplifies the problem because the existence of a correct future session is not guaranteed. For this setting, we then present a simpler algorithm that is restricted to safety specifications.

4.1 Parallel Trace Input Model

In short, we proceed as follows: First, since we know the number of traces, we can translate the HyperLTL formula to an equivalent LTL formula. For that formula, we construct a realizability monitor based on the LTL monitor described in [11]. The monitor is a parity game, which we use to detect minimal losing prefixes and to provide a valid continuation for the original HyperLTL formula.

Assume that the input model contains n traces. Let a HyperLTL formula $\forall \pi_1 \dots \forall \pi_k. \varphi$ over $\Sigma = I \dot\cup O$ be given, where φ is quantifier free. We construct an LTL formula φ^n_{LTL} over $\Sigma' = \{a_i \mid a \in \Sigma, 1 \leq i \leq n\}$ as follows:

$$\varphi^n_{\text{LTL}} \coloneqq \bigwedge_{i_1,\dots,i_k \in [1,n]} \varphi[\forall a \in AP : a_{\pi_1} \mapsto a_{i_1}, \dots, a_{\pi_k} \mapsto a_{i_k}]$$

The formula φ^n_{LTL} enumerates all possible combinations to choose k traces – one for each quantifier – from the set of n traces in the model. We use the notation $\varphi[\forall a \in AP : a_{\pi_1} \mapsto a_{i_1}, \dots, a_{\pi_k} \mapsto a_{i_k}]$ to indicate that in φ, atomic propositions with trace variables are replaced by atomic propositions indexed with one of the n traces. We define $I' = \{a_i \mid a \in I, 1 \leq i \leq n\}$ and O' analogously. Since n is known upfront, we only write φ_{LTL}.

Our algorithm exploits that for every LTL formula φ, there exists an equivalent parity game $\mathcal{G}_\varphi$ such that φ is realizable iff player P_0 is winning in the initial state with strategy σ_0 [13]. For a finite trace u, φ is realizable with prefix u iff the play induced by u ends in a state q that is in the winning region of player P_0. The algorithm to enforce the HyperLTL formula calls the following three procedures – depicted in Algorithm 1 – in the appropriate order.

Initialize: Construct φ_{LTL} and the induced parity game $\mathcal{G}_\varphi$. Solve the game $\mathcal{G}_\varphi$, i.e. compute the winning region for player P_0. If the initial state $q_0 \in V_0$ is losing, raise an error. Otherwise start monitoring in the initial state.

Algorithm 1 HyperLTL enforcement algorithm for the parallel input model.

```
1: procedure Initialize(ψ, n)
2:     φ_LTL := toLTL(ψ, n);
3:     (game, q_0) := toParity(φ_LTL);
4:     winR := SolveParity(game);
5:     if q_0 ∉ winR then
6:         raise error;
7:     return(game, winR, q_0);

8: procedure Enforce(game, lastq)
9:     sig := GetStrat(game, lastq);
10:    while true do
11:        o := sig(lastq);
12:        lastq := Move(game, lastq, o);
13:        output(o);
14:        i := getNextInput( );
15:        i_LTL := toLTL(i);
16:        lastq := Move(game, lastq, i_LTL);

17: procedure Monitor(game, winR, q)
18:    lastq := q;
19:    while true do
20:        o := getNextOutput( );
21:        o_LTL := toLTL(o);
22:        q := Move(game, lastq, o_LTL);
23:        if q ∉ winR then
24:            return(game, lastq);
25:        i := getNextInput( );
26:        i_LTL := toLTL(i);
27:        q := Move(game, q, i_LTL);
28:        lastq := q;
```

Monitor: Assume the game is currently in state $q \in V_0$. Get the next outputs $(o_1, \ldots, o_n) \in O^n$ produced by the n traces of the system and translate them to $o_{\mathrm{LTL}} \subseteq O'$ by subscripting them as described for formula φ_{LTL}. Move with o_{LTL} to the next state. This state is in V_1. Check if the reached state is still in the winning region. If not, it is a losing state, so we do not approve the system's output but let the enforcer take over and call Enforce on the last state. If the state is still in the winning region, we process the next inputs $(i_1, \ldots, i_n)$, translate them to i_{LTL}, and move with i_{LTL} to the next state in the game, again in V_0. While the game does not leave the winning region, the property is still realizable and the enforcer does not need to intervene.

Enforce: By construction, we start with a state $q \in V_0$ that is in the winning region, i.e., there is a positional winning strategy $\sigma : V_0 \rightarrow 2^{O'}$ for player P_0. Using this strategy, we output $\sigma(q)$ and continue with the next incoming input i_{LTL} to the next state in V_0. Continue with this strategy for any incoming input.

Correctness and Complexity. By construction, since we never leave the winning region, the enforced system fulfills the specification and the enforcer is sound. It is also transparent: As long as the prefix produced by the system is not losing, the enforcer does not intervene. The algorithm has triple exponential complexity in the number of traces n: The size of φ_{LTL} is exponential in n and constructing the parity game is doubly exponential in the size of φ_{LTL} [13]. Solving the parity game only requires quasi-polynomial time [34]. Note, however, that all of the above steps are part of the initialization. At runtime, the algorithm only follows the game arena. If the enforcer is only supposed to correct a single output and afterwards hand back control to the system, the algorithm could be easily adapted accordingly.

4.2 Sequential Trace Input Model

Deciding whether a prefix is losing in the sequential model is harder than in the parallel model. In the sequential model, strategies are defined w.r.t. the traces seen so far – they incrementally upgrade their knowledge with every new trace. In general, the question whether there exists a sound and transparent enforcement mechanism for universal HyperLTL specifications is undecidable.

Theorem 1. *In the sequential model, it is undecidable whether a HyperLTL formula φ from the universal fragment is enforceable.*

Proof. We encode the classic realizability problem of universal HyperLTL, which is undecidable [18], into the sequential model enforcement problem for universal HyperLTL. HyperLTL realizability asks if there exists a strategy $\sigma\colon (2^I)^* \to 2^O$ such that the set of traces constructed from every possible input sequence satisfies the formula φ, i.e. whether $\{(w[0] \cup \sigma(\epsilon)) \cdot (w[1] \cup \sigma(w[0])) \cdot (w[2] \cup \sigma(w[0..1])) \cdot \ldots \mid w \in (2^I)^\omega\}, \emptyset \models \varphi$. Let a universal HyperLTL formula φ over $\Sigma = I \dot\cup O$ be given. We construct $\psi \coloneqq \varphi \wedge \forall \pi. \forall \pi'. (\bigwedge_{o \in O} o_\pi \leftrightarrow o_{\pi'}) \ \mathcal{W} \ (\bigvee_{i \in I} i_\pi \not\leftrightarrow i_{\pi'})$. The universal HyperLTL formula ψ requires the strategy to choose the same outputs as long as the inputs are the same. The choice of the strategy must therefore be independent of earlier sessions, i.e., $\sigma(U, \epsilon)(s_I) = \sigma(U', \epsilon)(s_I)$ for all sets of traces U, U' and input sequences s_I. Any trace set that fulfills ψ can therefore be arranged in a traditional HyperLTL strategy tree branching on the inputs and labeling the nodes with the outputs. Assume the enforcer has to take over control after the first event when enforcing ψ. Thus, there is a sound and transparent enforcement mechanism for ψ iff φ is realizable. □

Finishing the Current Session. As the general problem is undecidable, we study the problem where the enforcer takes over control only for the rest of the current session. For the next session, the existence of a solution is not guaranteed. This approach is especially reasonable if we are confident that errors occur only sporadically. We adapt the algorithm presented for the parallel model. Let a HyperLTL formula $\forall \pi_1. \ldots \forall \pi_k. \varphi$ over $\Sigma = I \dot\cup O$ be given, where φ is quantifier free. As for the parallel model, we translate the formula into an LTL formula φ^n_{LTL}. We first do so for the first session with $n = 1$. We construct and solve the parity game for that formula, and use it to monitor the incoming events and to enforce the rest of the session if necessary. For the next session, we construct φ^n_{LTL} for $n = 2$ and add an additional conjunct encoding the observed trace t_1. The resulting formula induces a parity game that monitors and enforces the second trace. Like this, we can always enforce the current trace in relation to all traces seen so far. Algorithm 2 depicts the algorithm calling similar procedures as in Algorithm 1 (for which we therefore do not give any pseudo code). Initialize' is already given an LTL formula and, therefore, does not translate its input to LTL. Monitor' returns a tuple including the reason for its termination ('ok' when the trace finished and 'losing' when a losing prefix was detected). Additionally, the monitor returns the trace seen so far (not including the event that led to a losing prefix), which will be added to φ_{traces}. Enforce' enforces

Algorithm 2 HyperLTL enforcement algorithm for the sequential trace input model.

```
 1: procedure EnforceSequential(ψ)
 2:     n := 1;
 3:     φ_traces := true;
 4:     while true do
 5:         ψ_curr := toLTL(ψ, n) ∧ φ_traces;
 6:         (game, winR, q_0) := Initialize'(ψ_curr);
 7:         res := Monitor'(game, winR, q_0);
 8:         if res == ('ok', t) then
 9:             φ_traces := φ_traces ∧ toLTL(t);
10:         else if res = ('losing', t, (game, lastq)) then
11:             t' := Enforce'(game, lastq);
12:             φ_traces := φ_traces ∧ toLTL(t · t');
13:         n++;
```

the rest of the session and afterwards returns the produced trace, which is then encoded in the LTL formula (toLTL(t)).

Correctness and Complexity. Soundness and transparency follow from the fact that for the n-th session, the algorithm reduces the problem to the parallel setting with n traces, with the first $n-1$ traces being fixed and encoded into the LTL formula φ^n_{LTL}. We construct a new parity game from φ^n_{LTL} after each finished session. The algorithm is thus of non-elementary complexity.

Safety Specifications. If we restrict ourselves to formulas $\psi = \forall \pi_1 \ldots \forall \pi_k.\, \varphi$, where φ is a *safety* formula, we can improve the complexity of the algorithm. Note, however, that not every property with losing prefixes is a safety property: for the formula $\forall \pi. \square(o_\pi \rightarrow \diamondsuit i_\pi)$ with $o \in O$ and $i \in I$, any prefix with o set at some point is losing. However, the formula does not belong to the safety fragment. Given a safety formula φ, we can translate it to a safety game [25] instead of a parity game. The LTL formula we create with every new trace is built incrementally, i.e., with every finished trace we only ever add new conjuncts. With safety games, we can thus recycle the winning region from the game of the previous trace. The algorithm proceeds as follows. 1) Translate φ into an LTL formula φ^n_{LTL} for $n = 1$. 2) Build the safety game $\mathcal{G}^1_{\varphi_{\text{LTL}}}$ for φ^1_{LTL} and solve it. Monitor the incoming events of trace t_1 as before. Enforce the rest of the trace if necessary. 3) Once the session is terminated, generate the LTL formula $\varphi^2_{\text{LTL}} = \varphi^1_{\text{LTL}} \wedge \varphi^2_{\text{diff}}$. As φ^2_{LTL} is a conjunction of the old formula and a new conjunct φ^2_{diff}, we only need to generate the safety game $\mathcal{G}^2_{\text{diff}}$ and then build the product of $\mathcal{G}^2_{\text{diff}}$ with the winning region of $\mathcal{G}^1_{\varphi_{\text{LTL}}}$. We solve the resulting game and monitor (and potentially enforce) as before. The algorithm incrementally refines the safety game and enforces the rest of a session if needed. The construction recycles parts of the game computed for the previous session. We thus avoid the costly translation to a parity game for every new session. While constructing the safety game from the LTL specification has still doubly exponential complexity [25], solving safety games can be done in linear time [2].

Table 1: Enforcing symmetry in the Bakery protocol on pairs of traces. Times are given in seconds.

	random traces				symmetric traces			
$\|t\|$	avg	min	max	#enforced	avg	min	max	#enforced
500	0.003	0.003	0.003	0	0.013	0.008	0.020	10
1000	0.005	0.005	0.005	0	0.024	0.015	0.039	10
5.000	0.026	0.024	0.045	0	0.078	0.065	0.097	10
10.000	0.049	0.047	0.064	0	0.153	0.129	0.178	10

5 Experimental Evaluation

We implemented the algorithm for the parallel trace input model in our prototype tool REHyper[3], which is written in Rust. We use Strix [31] for the generation of the parity game. We determine the winning region and the positional strategies of the game with PGSolver [23]. All experiments ran on an Intel Xeon CPU E3-1240 v5 3.50 GHz, with 8 GB memory running Debian 10.6. We evaluate our prototype with two experiments. In the first, we enforce a non-trivial formulation of fairness in a mutual exclusion protocol. In the second, we enforce the information flow policy *observational determinism* on randomly generated traces.

5.1 Enforcing Symmetry in Mutual Exclusion Algorithms

Mutual exclusion algorithms like Lamport's bakery protocol ensure that multiple threads can safely access a shared resource. To ensure fair access to the resource, we want the protocol to be symmetric, i.e., for any two traces where the roles of the two processes are swapped, the grants are swapped accordingly. Since symmetry requires the comparison of two traces, it is a hyperproperty.

For our experiment, we used a Verilog implementation of the Bakery protocol [26], which has been proven to violate the following symmetry formulation [22]:

$$\forall \pi. \forall \pi'. (pc(0)_\pi = pc(1)_{\pi'} \land pc(1)_\pi = pc(0)_{\pi'})\, \mathcal{W}\, (pause_\pi = pause_{\pi'} \land \\ \mathtt{sym}(sel_\pi, sel_{\pi'}) \land \mathtt{sym}(break_\pi, break_{\pi'}) \land sel_\pi < 3 \land sel_{\pi'} < 3)\ .$$

The specification states that for any two traces, the program counters need to be symmetrical in the two processes as long as the processes are scheduled (*select*) and ties are broken (*break*) symmetrically. Both *pause* and $sel < 3$ handle further implementation details. The AIGER translation [22] of the protocol has 5 inputs and 46 outputs. To enforce the above formula, only 10 of the outputs are relevant. We enforced symmetry of the bakery protocol on simulated pairs of traces produced by the protocol. Table 1 shows our results for different trace lengths and trace generation techniques. We report the average runtime over 10 runs as well as minimal and maximal times along with the number of times

[3] REHyper is available at https://github.com/reactive-systems/REHyper

Table 2: Enforcing observational determinism. Times are given in seconds.

benchmark size				init time	0.5% bit flip probability				1% bit flip probability			
# i	# o	# t	|t|		avg	min	max	#enf'ed	avg	min	max	#enf'ed
1	1	3	10000	0.517	0.014	0.013	0.017	60	0.013	0.013	0.015	60
		8	10000	65.67	0.524	0.517	0.625	99	0.524	0.517	0.588	97
2	2	4	10000	0.869	0.025	0.024	0.030	73	0.032	0.031	0.043	77
		5	10000	21.189	0.038	0.037	0.041	90	0.038	0.037	0.042	86
3	3	2	10000	0.633	0.019	0.018	0.022	47	0.023	0.023	0.025	54
		4	5000	132.849	0.022	0.021	0.026	77	0.021	0.021	0.021	71
		4	10000		0.038	0.036	0.056	77	0.037	0.037	0.042	77
4	4	3	1000	43.885	0.010	0.008	0.015	71	0.009	0.008	0.018	68
		3	5000		0.023	0.021	0.033	72	0.022	0.021	0.025	78
		3	10000		0.038	0.037	0.050	76	0.038	0.037	0.041	75

the enforcer needed to intervene. The symmetry assumptions are fairly specific and are unlikely to be reproduced by random input simulation. In a second experiment, we therefore generated pairs of symmetric traces. Here, the enforcer had to intervene every time, which produced only a small overhead.

The required game was constructed and solved in 313 seconds. For sets of more than two traces, the construction of the parity game did not return within two hours. The case study shows that the tool performs without significant overhead at runtime and can easily handle very long traces. The bottleneck is the initial parity game construction and solving.

5.2 Enforcing Observational Determinism

In our second experiment, we enforced observational determinism, given as the HyperLTL formula $\forall \pi. \forall \pi'. (o_\pi \leftrightarrow o_{\pi'}) \mathcal{W} (i_\pi \nleftrightarrow i_{\pi'})$. The formula states that for any two execution traces, the observable outputs have to agree as long as the observable inputs agree. Observational determinism is a prototypical information-flow policy used in many experiments and case studies for HyperLTL (e.g. [18,22,4]). We generated traces using the following scalable generation scheme: At each position, each input and output bit is flipped with a certain probability (0.5% or 1%). This results in instances where observational determinism randomly breaks. Table 2 shows our results. Each line corresponds to 100 randomly generated instances of the given size (number of inputs/outputs and traces, and length of the sessions). We report the initialization time that is needed to generate and solve the parity game. Furthermore, we report average, minimal, and maximal enforcement time as well as the number of instances where the enforcer intervened. All times are reported in seconds. The bottleneck is again the time needed to construct and solve the parity game. At runtime, which is the crucial aspect, the enforcer performs efficiently. The higher bit flip probability did not lead to more enforcements: For traces of length 10000, the probability that the enforcer intervenes is relatively high already at a bit flip probability of 0.5%.

6 Conclusion

We studied the runtime enforcement problem for hyperproperties. Depending on the trace input model, we showed that the enforcement problem boils down to detecting losing prefixes and solving a custom synthesis problem. For both input models, we provided enforcement algorithms for specifications given in the universally quantified fragment of the temporal hyperlogic HyperLTL. While the problem for the sequential trace input model is in general undecidable, we showed that enforcing HyperLTL specifications becomes decidable under the reasonable restriction to only finish the current session. For the parallel model, we provided an enforcement mechanism based on parity game solving. Our prototype tool implements this algorithm for the parallel model. We conducted experiments on two case studies enforcing complex HyperLTL specifications for reactive systems with the parallel model. Our results show that once the initial parity game solving succeeds, our approach has only little overhead at runtime and scales to long traces.

References

1. Agrawal, S., Bonakdarpour, B.: Runtime Verification of k-Safety Hyperproperties in HyperLTL. In: CSF 2016
2. Beeri, C.: On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases. ACM Trans. Database Syst. **5**(3), 241–259 (1980)
3. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield Synthesis: Runtime Enforcement for Reactive Systems. CoRR **abs/1501.02573** (2015)
4. Bonakdarpour, B., Finkbeiner, B.: The Complexity of Monitoring Hyperproperties. In: CSF 2018
5. Bozzelli, L., Maubert, B., Pinchinat, S.: Unifying Hyper and Epistemic Temporal Logics. In: FSSCS 2015
6. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: CSF 2008
7. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal Logics for Hyperproperties. In: POST 2014
8. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J.: The Hierarchy of Hyperlogics. In: LICS 2019
9. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying Hyperliveness. In: CAV 2019
10. Devriese, D., Piessens, F.: Noninterference through Secure Multi-execution. In: S&P 2010
11. Ehlers, R., Finkbeiner, B.: Monitoring Realizability. In: RV 2011
12. Erlingsson, U., Schneider, F.B.: SASI Enforcement of Security Policies: A Retrospective. In: NSPW 1999
13. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From LTL and Limit-Deterministic Büchi Automata to Deterministic Parity Automata. In: TACAS 2017
14. Falcone, Y.: You Should Better Enforce Than Verify. In: RV 2010
15. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? Int. J. Softw. Tools Technol. Transf. **14**(3), 349–382 (2012)
16. Falcone, Y., Mounier, L., Fernandez, J.C., Richier, J.L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. FMSD **38** (2011)

17. Falcone, Y., Pinisetty, S.: On the Runtime Enforcement of Timed Properties. In: RV 2019
18. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesizing Reactive Systems from Hyperproperties. In: CAV 2018
19. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring Hyperproperties. FMSD 2019
20. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: A Runtime Verification Tool for Temporal Hyperproperties. In: TACAS 2018
21. Finkbeiner, B., Hahn, C., Torfah, H.: Model Checking Quantitative Hyperproperties. In: CAV 2018
22. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for Model Checking HyperLTL and HyperCTL*. In: CAV 2015
23. Friedmann, O., Lange, M.: The PGSolver Collection of Parity Game Solvers Version 3 (2010)
24. Hahn, C., Stenger, M., Tentrup, L.: Constraint-Based Monitoring of Hyperproperties. In: TACAS 2019
25. Kupferman, O., Vardi, M.Y.: Model Checking of Safety Properties. In: CAV 1999
26. Lamport, L.: A New Solution of Dijkstra's Concurrent Programming Problem. Commun. ACM **17**(8), 453–455 (1974)
27. Li, X., Dunn, J., Salins, D., Zhou, G., Zhou, W., Rose, S.M.S.F., Perelman, D., Colbert, E., Runge, R., Rego, S., et al.: Digital Health: Tracking Physiomes and Activity Using Wearable Biosensors Reveals Useful Health-Related Information. PLoS Biology (2017)
28. Ligatti, J., Bauer, L., Walker, D.: Run-Time Enforcement of Nonsafety Policies. ACM Trans. Inf. Syst. Secur. **12**(3) (Jan 2009)
29. Luo, Q., Roundefinedu, G.: EnforceMOP: A Runtime Property Enforcement System for Multithreaded Programs. In: ISSTA 2013
30. McLean, J.: Proving Noninterference and Functional Correctness Using Traces. Journal of Computer Security **1**(1), 37–58 (1992)
31. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit Reactive Synthesis Strikes Back! In: CAV 2018
32. Ngo, M., Massacci, F., Milushev, D., Piessens, F.: Runtime Enforcement of Security Policies on Black Box Reactive Programs. In: POPL 2015
33. Norman, G., Shmatikov, V.: Analysis of Probabilistic Contract Signing. JCS 2006
34. Parys, P.: Parity Games: Zielonka's Algorithm in Quasi-Polynomial Time. In: MFCS 2019
35. Pnueli, A.: The Temporal Logic of Programs. In: SFCS 1977
36. Renard, M., Falcone, Y., Rollet, A., Jéron, T., Marchand, H.: Optimal Enforcement of (Timed) Properties with Uncontrollable Events. MSCS 2019
37. Roscoe, A.W.: CSP and determinism in security modelling. In: S&P 1995
38. Schneider, F.B.: Enforceable Security Policies. ACM Trans. Inf. Syst. Secur. 2000
39. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-Box Monitoring of Hyperproperties. In: FM 2019
40. Wu, M., Zeng, H., Wang, C.: Synthesizing Runtime Enforcer of Safety Properties Under Burst Error. In: NFM 2016
41. Zdancewic, S., Myers, A.C.: Observational Determinism for Concurrent Program Security. In: CSFW-16 2003