





Software Verification of Hyperproperties Beyond k -Safety

Raven Beutner^(✉)  and Bernd Finkbeiner 

CISPA Helmholtz Center for Information Security,
Saarbrücken, Germany
{raven.beutner, finkbeiner}@cispa.de



Abstract. Temporal hyperproperties are system properties that relate multiple execution traces. For (finite-state) hardware, temporal hyperproperties are supported by model checking algorithms, and tools for general temporal logics like HyperLTL exist. For (infinite-state) software, the analysis of temporal hyperproperties has, so far, been limited to k -safety properties, i.e., properties that stipulate the absence of a bad interaction between any k traces. In this paper, we present an automated method for the verification of $\forall^k\exists^l$ -safety properties in infinite-state systems. A $\forall^k\exists^l$ -safety property stipulates that for any k traces, there *exist* l traces such that the resulting $k + l$ traces do not interact badly. This combination of universal and existential quantification enables us to express many properties beyond k -safety, including, for example, generalized non-interference or program refinement. Our method is based on a strategy-based instantiation of existential trace quantification combined with a program reduction, both in the context of a fixed predicate abstraction. Notably, our framework allows for mutual dependence of strategy and reduction.

Keywords: Hyperproperties · HyperLTL · Infinite-state systems · Predicate abstraction · Hyperliveness · Verification · Program reduction

1 Introduction

Hyperproperties are system properties that relate multiple execution traces of a system [22] and commonly arise, e.g., in information-flow policies [35], the verification of code optimizations [6], and robustness of software [19]. Consequently, many methods for the automated verification of hyperproperties have been developed [27, 39–41]. Almost all previous approaches verify a class of hyperproperties called k -safety, i.e., properties that stipulate the absence of a bad interaction between any k traces in the system. For example, we can express a simple form of non-interference as a 2-safety property by stating that any *two* traces that agree on the low-security inputs should produce the same observable output.

The vast landscape of hyperproperties does, however, stretch far beyond k -safety. The overarching limitation of k -safety (or, more generally, of hypersafety [22]) is an implicit *universal* quantification over all executions. By contrast, many

properties of interest, ranging from applications in information-flow control to robust cleanness, require a combination of universal and existential quantification. For example, consider the reactive program in Fig. 1, where $\star_{\mathbb{N}}$ denotes a nondeterministic choice of a natural number. We assume that h , l , and o are a high-security input, a low-security input, and a low-security output, respectively. This program violates the simple 2-safety non-interference property given above as the non-determinism influences the output. Nevertheless, the program is “secure” in the sense that an attacker that observes low-security inputs and outputs cannot deduce information about the high-security input. To capture this formally, we use a relaxed notion of non-interference, in the literature often referred to as generalized non-interference (GNI) [35]. We can, informally, express GNI in a temporal logic as follows:

$$\forall \pi. \forall \pi'. \exists \pi''. \Box (o_{\pi} = o_{\pi''} \wedge l_{\pi} = l_{\pi''} \wedge h_{\pi'} = h_{\pi''})$$

This property requires that for any two traces π, π' , there exists some trace π'' that, globally, agrees with the low-security inputs and outputs on π but the high-security inputs on π' . Phrased differently, any observation on the low-security input-output behavior is compatible with every possible high-security input. The program in Fig. 1 satisfies GNI. Crucially, GNI is no longer a hypersafety property (and, in particular, no k -safety property for any k) as it requires a combination of universal and *existential* quantification.

1.1 Verification Beyond k -Safety

Instead, GNI falls in the general class of $\forall^* \exists^*$ -safety properties. Concretely, a $\forall^k \exists^l$ -safety property (using k universal and l existential quantifiers) stipulates that for any k traces, there exist l traces such that the resulting $k + l$ traces do not interact badly. k -safety properties are the *special case* where $l = 0$. We study the verification of such properties in infinite-state systems arising, e.g., in software. In contrast to k -safety, where a broad range of methods has been developed [10, 27, 39–41], no method for the automated verification of *temporal* $\forall^* \exists^*$ properties in infinite-state systems exists (we discuss related approaches in Sect. 8).

Our novel verification method is based on a game-based reading of existential quantification *combined* with the search for a program reduction. The game-based reading of existential quantification instantiates existential trace quantification with an explicit strategy and constitutes the first practicable method for the verification of $\forall^* \exists^*$ -properties in finite-state systems [23]. Program reductions are a well-established technique to align executions of independent program fragments (such as the individual program copies in a self-composition) to obtain proofs with easier invariants [27, 34, 39].

```

repeat
  readInput( $h, l$ )
  if  $h > l$  then
     $o \leftarrow l + \star_{\mathbb{N}}$ 
  else
     $x \leftarrow \star_{\mathbb{N}}$ 
    if  $x \geq l$  then
       $o \leftarrow x$ 
    else
       $o \leftarrow l$ 

```

Fig. 1. An example program is depicted.

So far, both techniques are limited to their respective domain, i.e., the game-based approach has only been applied to finite-state systems and synchronous specifications, and reductions have (mostly) been used for the verification of k -safety. We combine both techniques yielding an effective (and first) verification technique for hyperproperties beyond k -safety in infinite-state systems arising in software. Notably, our search for reduction and strategy-based instantiation of existential quantification is *mutually dependent*, i.e., a particular strategy might depend on a particular reduction and vice versa.

1.2 Contributions and Structure

The starting point of our work is a new temporal logic called *Observation-based HyperLTL* (OHyperLTL for short). Our logic extends the existing hyperlogic HyperLTL [21] with capabilities to reason about asynchronous properties (i.e., properties where the individual traces are traversed at different speeds), and to specify properties using assertions from arbitrary background theories (to reason about the infinite domains encountered in software) (Sect. 4).

To automatically verify $\forall^k \exists^l$ OHyperLTL properties, we combine program reductions with a strategy-based instantiation of existential quantification, both in the context of a fixed predicate abstraction. To facilitate this combination, we first present a game-based approach that automates the search for a reduction. Concretely, we construct an abstract game where a winning strategy for the verifier directly corresponds to a reduction with accompanying proof. As a side product, our game-based interpretation simplifies the search for a reduction in a given predicate abstraction as, e.g., studied by Shemer et al. [39] (Sect. 5).

Our strategic (game-based) view on reductions allows us to combine them with a game-based instantiation of existential quantification. Here, we view the existentially quantified traces as being constructed by a strategy that, iteratively, reacts to the universally quantified traces. As we phrase both the search for a reduction and the search for existentially quantified traces as a game, we can frame the search for both as a combined abstract game. We prove the soundness of our approach, i.e., a winning strategy for the verifier constitutes both a strategy for the existentially quantified traces and accompanying (mutually dependent) reduction. Despite its finite nature, constructing the abstract game is expensive as it involves many SMT queries. We propose an inner refinement loop that determines the winner of the game (without constructing it explicitly) by computing iterative approximations (Sect. 6).

We have implemented our verification approach in a prototype tool called HyPA (short for **H**yperproperty **V**erification with **P**redicate **A**bstraction) and evaluate HyPA on k -safety properties (that can already be handled by existing methods) and on $\forall^* \exists^*$ -safety benchmarks that cannot be handled by any existing tool (Sect. 7).

Contributions. In short, our contributions include the following:

- We propose a temporal hyperlogic that can specify asynchronous hyperproperties in infinite-state systems;

- We propose a game-based interpretation of a reduction (improving and simplifying previous methods for k -safety [39]);
- We combine a strategy-based instantiation of existentially quantified traces with the search for a reduction. This yields a flexible (and first) method for the verification of temporal $\forall^*\exists^*$ properties. We propose an iterative method to solve the abstract game that avoids an expensive explicit construction;
- We provide and evaluate a prototype implementation of our method.

2 Overview: Reductions and Quantification as a Game

Our verification approach hinges on the observation that we can express both a reduction and existential trace quantification as a game. In this section, we provide an overview of our game-based interpretations. We begin by outlining our game-based reading of a reduction (illustrating this in the simpler case of k -safety) in Sect. 2.1 and then extend this to include a game-based interpretation of existential quantification in Sect. 2.2.

2.1 Reductions as a Game

Consider the two programs in Fig. 2 and the specification that both programs produce the same output (on initially identical values for x). We can formalize this in our logic OHyperLTL (formally defined in Sect. 4) as follows:

$$\forall^{P1} \pi_1 : (pc = 2). \forall^{P2} \pi_2 : (pc = 2). (x_{\pi_1} = x_{\pi_2}) \rightarrow \Box(x_{\pi_1} = x_{\pi_2})$$

The property states that for all traces π_1 in P1 and π_2 in P2 the LTL specification $(x_{\pi_1} = x_{\pi_2}) \rightarrow \Box(x_{\pi_1} = x_{\pi_2})$ holds (where x_π refers to the value of x on trace π). Additionally, the observation formula $pc = 2$ marks the positions at which the LTL property is evaluated: We only observe a trace at steps where $pc = 2$ (i.e., where the program counter is at the output position).

The verification of our property involves reasoning about two copies of our system (in this case, one of P1 and one of P2) on *disjoint* state spaces. Consequently, we can interleave the statements of both programs (between two observation points) without affecting the behavior of the individual copies. We refer to each interleaving of both copies as a *reduction*. The choice of a reduction drastically influences the complexity of the needed invariants [27, 34, 39]. Given an initial abstraction of the system [30, 39], we aim to discover a suitable reduction *automatically*. Our first observation is that we can phrase the search for a reduction as a game as follows: In each step, the verifier decides on a *scheduling* (i.e., a non-empty subset $M \subseteq \{1, 2\}$) that indicates which of the copies should take a step (i.e., $i \in M$ iff copy i should make a program step). Afterward, the refuter can choose an abstract successor state compatible with that scheduling, after which the process repeats. This naturally defines a finite-state two-player safety game that we can solve efficiently.¹ If the verifier wins, a winning strategy

¹ The LTL specification is translated to a symbolic safety automaton that moves alongside the game. For sake of readability, we omitted the automaton from the following discussion.

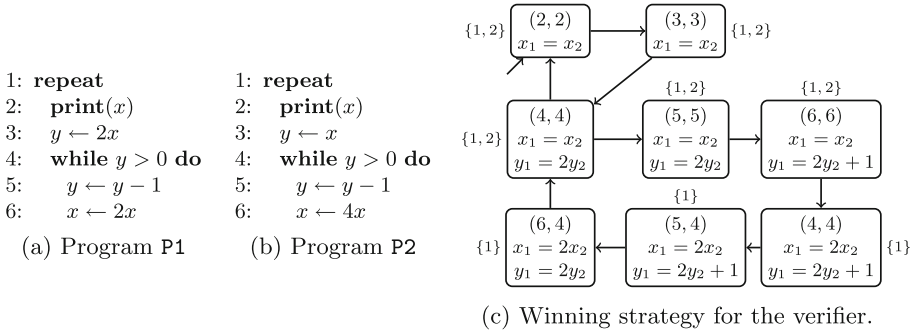


Fig. 2. Two output-equivalent programs P1 and P2 are depicted in Fig. 2a and 2b. In Fig. 2c a possible winning strategy for the verifier is given. Each abstract state contains the value of the program counter of both copies (given as the pair at the top) and the predicates that hold in that state. For sake of readability we omit the trace variables and write, e.g., x_1 for x_{π_1} . We mark the initial state with an incoming arrow. The outer label at each state gives the scheduling $M \subseteq \{1, 2\}$ chosen by the strategy in that state.

directly corresponds to a reduction and accompanying inductive invariant for the safety property within the given abstraction.

For our example, we give (parts of) a possible winning strategy in Fig. 2c. In each abstract state, the strategy chooses a scheduling (written next to the state), and all abstract states compatible with that scheduling are listed as successors. Note that whenever the program counter is (2, 2) (i.e., both programs are at their output position), it holds that $x_1 = x_2$ (as required). The example strategy schedules in lock-step for the most part (by choosing $M = \{1, 2\}$) but lets P1 take the inner loop *twice*, thereby maintaining the linear invariants $x_1 = x_2$ and $y_1 = 2y_2$. In particular, the resulting reduction is property-based [39] as the scheduling is based on the current (abstract) state. Note that the program cannot be verified with only linear invariants in a sequential or parallel (lock-step) reduction.

2.2 Beyond k -Safety: Quantification as a Game

We build upon this game-based interpretation of a reduction to move beyond k -safety. As a second example, consider the two programs Q1 and Q2 in Fig. 3, where \star_τ denotes a nondeterministic choice of type $\tau \in \{\mathbb{N}, \mathbb{B}\}$. We wish to check that Q1 refines Q2, i.e., all output behavior of Q1 is also possible in Q2. We can express this in our logic as follows:

$$\forall^{Q1} \pi_1 : (pc = 2). \exists^{Q2} \pi_2 : (pc = 2). \Box (a_{\pi_1} = a_{\pi_2})$$

The property states that for every trace π_1 in Q1 there *exists* a trace π_2 in Q2 that outputs the same value. The quantifiers range over infinite traces of variable assignments (with infinite domains), making a direct verification of the

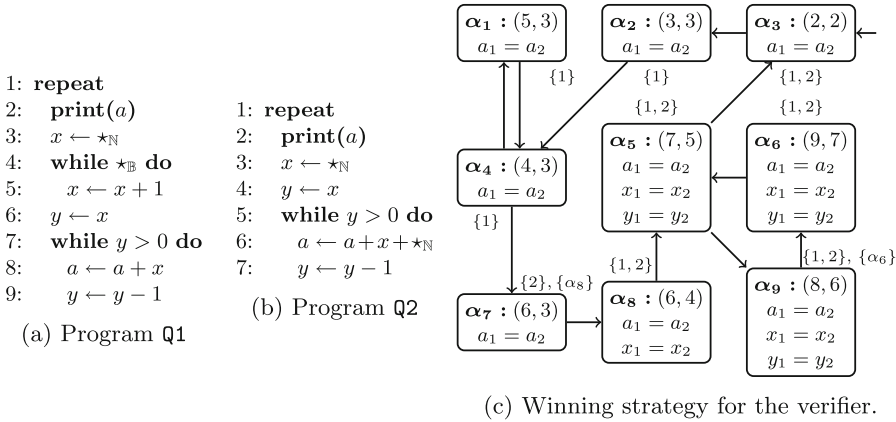


Fig. 3. Two programs Q1 and Q2 are given in Fig. 3a and 3b. In Fig. 3c a possible winning strategy for the verifier is depicted. The outer label gives the scheduling $M \subseteq \{1, 2\}$ and, if applicable, the restriction chosen by the witness strategy.

quantifier alternation challenging. In contrast to alternation-free formulas, we cannot reduce the verification to verification on a self composition [8, 28]. Instead, we adopt (yet another) game-based interpretation by viewing the existentially quantified traces as being resolved by a *strategy* (called the witness strategy) [23]. That is, instead of trying to find a witness traces π_2 in Q2 when given the *entire* trace π_1 , we interpret the $\forall\exists$ property as a game between verifier and refuter. The refuter moves through the state space of Q1 (thereby producing a trace π_1), and the verifier reacts to each move by choosing a successor in the state space of Q2 (thereby producing a trace π_2). If the verifier can assure that the resulting traces π_1, π_2 satisfy $\Box(a_{\pi_1} = a_{\pi_2})$, the $\forall\exists$ property holds. However, this game-based interpretation fails in many instances. There might exist a witness trace π_2 , but the trace cannot be produced by a witness strategy as it requires knowledge of *future* moves of the refuter. Let us discuss this on the example programs in Fig. 3. A simple (informal) solution to construct a witness trace π_2 (when given the entire π_1) would be to guarantee that in Q2:4 (meaning location 4 of Q2) and line Q1:6 the value of x in both programs agrees (i.e., $x_1 = x_2$ holds) and then simply resolve the nondeterminism at Q2:6 with 0. However, to follow this idea, the witness strategy for the verifier, when at Q2:3, would need to know the future value of x_1 when Q1 is at location Q1:6.

Our insight in this paper is that we can turn the strategy-based interpretation of the witness trace π_2 into a useful verification method by *combining* it with a program reduction. As we express both searches strategically, we can phrase the combined search as a combined game. In particular, both the reduction and the witness strategy are controlled by the verifier and can thus *collaborate*. In the resulting game, the verifier chooses a scheduling (as in Sect. 2.1) and, additionally, whenever the existentially quantified copy is scheduled, the verifier also decides on the successor state of that copy. We depict a possible winning strat-

egy in Fig. 3c. This strategy formalizes the interplay of reduction and witness strategy. Initially, the verifier only schedules $\{1\}$ until Q1 has reached program location Q1:6 (at which point the value of x is fixed). Only then does the verifier schedule $\{2\}$, at which point the witness strategy can decide on a successor state for Q2. In our case, the strategy chooses a value for x such that $x_1 = x_2$ holds. As we work in an abstraction of the actual system, we formalize this by restricting the abstract successor states. In particular, in state α_7 the verifier schedules $\{2\}$ and simultaneously restricts the successors to $\{\alpha_8\}$ (i.e., the abstract state where $x_1 = x_2$ holds), even though abstract state $[(6, 4), a_1 = a_2, x_1 \neq x_2]$ is also a valid successor under scheduling $\{2\}$. We formalize when a restriction is valid in Sect. 6. The resulting strategy is winning and therefore denotes both a reduction *and* witness strategy for the existentially quantified copy. Importantly, both reduction and witness strategy are mutually dependent. Our tool HyPA is able to verify both properties (in Fig. 2 and Fig. 3) in a matter of a few seconds (cf. Sect. 7).

3 Preliminaries

We begin by introducing basic preliminaries, including our basic model of computation and background on (finite-state) safety games.

Symbolic Transition Systems. We assume some fixed underlying first-order theory. A *symbolic transition system* (STS) is a tuple $\mathcal{T} = (X, \text{init}, \text{step})$ where X is a finite set of variables (possibly sorted), init is a formula over X describing all initial states, and step is a formula over $X \uplus X'$ (where $X' := \{x' \mid x \in X\}$ is the set of primed variables) describing the transitions of the system. A concrete state μ in \mathcal{T} is an assignment to the variables in X . We write μ' for the assignment over X' given by $\mu'(x') := \mu(x)$. A trace in \mathcal{T} is an infinite sequence of assignment $\mu_0 \mu_1 \cdots$ such that $\mu_0 \models \text{init}$ and for every $i \in \mathbb{N}$, $\mu_i \uplus \mu'_{i+1} \models \text{step}$. We write $\text{Traces}(\mathcal{T})$ for the set of all traces in \mathcal{T} . We can naturally interpret programs as STS by making the program counter explicit.

Formula Transformations. For the remainder of this paper, we fix the set of system variables X . We also fix a finite set of trace variables $\mathcal{V} = \{\pi_1, \dots, \pi_k\}$. For a trace variable $\pi \in \mathcal{V}$ we define $X_\pi := \{x_\pi \mid x \in X\}$ and write \vec{X} for $X_{\pi_1} \cup \dots \cup X_{\pi_k}$. For a formula θ over X , we define $\theta_{\langle \pi \rangle}$ as the formula over X_π obtained by replacing every variable x with x_π . Similarly, we define k fresh disjoint copies $\vec{X}' = X'_{\pi_1} \cup \dots \cup X'_{\pi_k}$ (where $X'_\pi := \{x'_\pi \mid x \in X\}$). For a formula θ over \vec{X} , we define $\theta^{(\prime)}$ as the formula over \vec{X}' obtained by replacing every variable x_π with x'_π .

Safety Games. A *safety game* is a tuple $\mathcal{G} = (S_{\text{SAFE}}, S_{\text{REACH}}, S_0, T, B)$ where $S = S_{\text{SAFE}} \uplus S_{\text{REACH}}$ is a set of game states, $S_0 \subseteq S$ a set of initial states, $T \subseteq S \times S$ a transition relation, and $B \subseteq S$ a set of bad states. We assume that for every $s \in S$ there exists at least one s' with $(s, s') \in T$. States in S_{SAFE} are controlled by

player **SAFE** and those in S_{REACH} by player **REACH**. A play is an infinite sequence of states $s_0 s_1 \dots$ such that $s_0 \in S_0$, and $(s_i, s_{i+1}) \in T$ for every $i \in \mathbb{N}$. A positional strategy σ for player $p \in \{\text{SAFE}, \text{REACH}\}$ is a function $\sigma : S_p \rightarrow S$ such that $(s, \sigma(s)) \in T$ for every $s \in S_p$. A play $s_0 s_1 \dots$ is compatible with strategy σ for player p if $s_{i+1} = \sigma(s_i)$ whenever $s_i \in S_p$. The safety player wins \mathcal{G} if there is a strategy σ for **SAFE** such that all σ -compatible plays never visit a state in B . In particular, **SAFE** needs to win from *all* initial states.

4 Observation-Based HyperLTL

In this section, we present OHyperLTL (short for observation-based HyperLTL). Our logic builds upon HyperLTL [21], which itself extends linear-time temporal logic (LTL) with explicit trace quantification. In OHyperLTL, we include predicates from the background theory (to reason about infinite variable domains) and explicit observations (to express asynchronous properties). Formulas in OHyperLTL are given by the following grammar:²

$$\begin{aligned} \varphi &:= \forall \pi : \xi. \varphi \mid \exists \pi : \xi. \varphi \mid \phi \\ \phi &:= \theta \mid \neg \phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc \phi \mid \phi_1 \mathcal{U} \phi_2 \end{aligned}$$

Here $\pi \in \mathcal{V}$ is a trace variable, θ is a formula over \vec{X} , and ξ is a formula over X (called the observation formula). For ease of notation, we assume that all variables in \mathcal{V} occur in the quantifier prefix *exactly* once. We use the standard Boolean connectives \wedge , \rightarrow , \leftrightarrow , and constants \top , \perp , as well as the derived LTL operators eventually $\diamond \phi := \top \mathcal{U} \phi$, and globally $\square \phi := \neg \diamond \neg \phi$.

Semantics. A trace t is an infinite sequence $\mu_0 \mu_1 \dots$ of assignments to X . For $i \in \mathbb{N}$, we write $t(i)$ to denote the i th value in t . A trace assignment Π is a partial mapping of trace variables in \mathcal{V} to traces. Given a trace assignment Π and $i \in \mathbb{N}$, we define $\Pi(i)$ to be the assignment to \vec{X} given by $\Pi(i)(x_\pi) := \Pi(\pi)(i)(x)$, i.e., the value of x_π is the value of x on the trace assigned to π . For the LTL body of an OHyperLTL formula, we define:

$$\begin{aligned} \Pi, i \models \theta & \quad \text{iff} \quad \Pi(i) \models \theta \\ \Pi, i \models \neg \phi & \quad \text{iff} \quad \Pi, i \not\models \phi \\ \Pi, i \models \phi_1 \wedge \phi_2 & \quad \text{iff} \quad \Pi, i \models \phi_1 \text{ and } \Pi, i \models \phi_2 \\ \Pi, i \models \bigcirc \phi & \quad \text{iff} \quad \Pi, i + 1 \models \phi \\ \Pi, i \models \phi_1 \mathcal{U} \phi_2 & \quad \text{iff} \quad \exists j \geq i. \Pi, j \models \phi_2 \text{ and } \forall i \leq k < j. \Pi, k \models \phi_1 \end{aligned}$$

The distinctive feature of OHyperLTL over HyperLTL are the explicit observations. Given an observation formula ξ and trace t , we say that ξ is a *valid*

² For the examples in Sect. 2, we additionally annotated quantifiers with an STS if we want to reason about different STSs within the same formula. In the following, we assume that all quantifiers range over traces in the same STS to simplify notation.

observation on t (written $\text{valid}(t, \xi)$) if there are infinitely many $i \in \mathbb{N}$ such that $t(i) \models \xi$. If $\text{valid}(t, \xi)$ holds, we write $\langle t \rangle_\xi$ for the trace obtained by projecting on those positions i where $t(i) \models \xi$, i.e., $\langle t \rangle_\xi(i) := t(j)$ where j is the i th index that satisfies ξ . Given a set of traces \mathbb{T} we resolve trace quantification as follows:

$$\begin{aligned} \Pi \models_{\mathbb{T}} \phi & \quad \text{iff} \quad \Pi, 0 \models \phi \\ \Pi \models_{\mathbb{T}} \forall \pi : \xi. \varphi & \quad \text{iff} \quad \forall t \in \{t \in \mathbb{T} \mid \text{valid}(t, \xi)\}. \Pi[\pi \mapsto \langle t \rangle_\xi] \models_{\mathbb{T}} \varphi \\ \Pi \models_{\mathbb{T}} \exists \pi : \xi. \varphi & \quad \text{iff} \quad \exists t \in \{t \in \mathbb{T} \mid \text{valid}(t, \xi)\}. \Pi[\pi \mapsto \langle t \rangle_\xi] \models_{\mathbb{T}} \varphi \end{aligned}$$

The semantics mostly agrees with that of HyperLTL [21] but projects each trace to the positions where the observation holds. Given an STS \mathcal{T} and OHyperLTL formula φ , we write $\mathcal{T} \models \varphi$ if $\emptyset \models_{\text{Traces}(\mathcal{T})} \varphi$ where \emptyset is the empty assignment.

The Power of Observations. The explicit observations in OHyperLTL facilitate the specification of asynchronous hyperproperties, i.e., properties where traces are traversed at different speeds. For the example in Sect. 2.1, the explicit observations allow us to compare the output of both programs even though the actual step at which the output occurs (in a synchronous semantics) differs between both programs (as P1 takes the inner loop twice as often as P2). As the observations are part of the specification, we can model a broad spectrum of properties ranging, e.g., from timing-insensitive properties (by placing observations only at output locations) to timing-sensitive specifications [29] (by placing observations at closer intervals). Functional (opposed to temporal) k -safety properties specified by pre-and postcondition [10, 39, 41] can easily be encoded as \forall^k -OHyperLTL properties by placing observations at the start and end of each program. By setting $\xi = \top$, i.e., observing *every* step, we can express synchronous properties. OHyperLTL thus subsumes HyperLTL.

Finite-State Model Checking. Many mechanisms used to express asynchronous hyperproperties render finite-state model checking undecidable [9, 17, 31]. In contrast, the simple mechanism used in OHyperLTL maintains decidable finite-state model checking. Detailed proofs can be found in the full version [15].

Theorem 1. *Assume an STS \mathcal{T} with finite variable domains and decidable background theory and an OHyperLTL formula φ . It is decidable if $\mathcal{T} \models \varphi$.*

Proof Sketch. Under the assumptions, we can view \mathcal{T} as an explicit (instead of symbolic) finite-state transition system. Given an observation formula ξ we can effectively compute an explicit finite-state system \mathcal{T}' such that $\text{Traces}(\mathcal{T}') = \{\langle t \rangle_\xi \mid t \in \text{Traces}(\mathcal{T}) \wedge \text{valid}(t, \xi)\}$. This reduces OHyperLTL model checking on \mathcal{T} to HyperLTL model checking on \mathcal{T}' , which is decidable [28]. \square

Note that for infinite-state (symbolic) systems, we cannot effectively compute \mathcal{T}' as in the proof of Theorem 1. In fact, there may not even exist a system \mathcal{T}' with the desired property that is expressible in the same background theory.

The finite-state result in Theorem 1 is of little relevance for the present paper. Nevertheless, it indicates that our logic is well suited for verification of infinite-state (software) systems as the (inevitable) undecidability stems from the infinite domains in software programs and not already from the logic itself.

Safety. In this paper, we assume that the hyperproperty is temporally safe [12], i.e., the temporal body of any OHyperLTL formula denotes a *safety property*. Note that, as we support quantifier alternation, we can still express hyperliveness properties [22, 23]. For example, GNI is both temporally safe and hyperliveness. We model the body of a formula by a symbolic safety automaton [24], which is a tuple $\mathcal{A} = (Q, q_0, \delta, B)$ where Q is a finite set of states, $q_0 \in Q$ the initial state, $B \subseteq Q$ a set of bad-states, and δ a finite set of automaton edges of the form (q, θ, q') where $q, q' \in Q$ are states and θ is a formula over \vec{X} . Given a trace t over assignments to \vec{X} , a run of \mathcal{A} on t is an infinite sequence of states $q_0 q_1 \dots$ (starting in q_0) such that for every i , there exists an edge $(q_i, \theta_i, q_{i+1}) \in \delta$ such that $t(i) \models \theta_i$. A word is accepted by \mathcal{A} if it has *no* run that visits a state in B . The automaton is *deterministic* if for every $q \in Q$ and every assignments μ to \vec{X} , there exists exactly one edge $(q, \theta, q') \in \delta$ with $\mu \models \theta$.

5 Reductions as a Game

After having defined our temporal logic, we turn our attention to the automatic verification of OHyperLTL formulas on STSs. In this section, we begin by formalizing our game-based interpretation of a reduction. To illustrate this, we consider \forall^k OHyperLTL formulas, which, as the body of the formula is a safety property, always denote k -safety properties.

Predicate Abstraction. Our search for a reduction is based in the scope of a fixed predicate abstraction [30, 33], i.e., we abstract our system by keeping track of the truth value of a few selected predicates that (ideally) identify properties that are relevant to prove the property in question. Let $\mathcal{T} = (X, \text{init}, \text{step})$ be an STS and let $\varphi = \forall \pi_1 : \xi_1 \dots \forall \pi_k : \xi_k. \phi$ be the (k -safety) OHyperLTL we wish to verify. Let $\mathcal{A}_\phi = (Q_\phi, q_{\phi,0}, \delta_\phi, B_\phi)$ be a deterministic safety automaton for ϕ . A *relational* predicate p is a formula over \vec{X} that identifies a property of the combined state space of k system copies. Let $\mathcal{P} = \{p_1, \dots, p_n\}$ be a finite set of relational predicates. We say a formula over \vec{X} is *expressible in \mathcal{P}* if it is equivalent to a boolean combination of the predicates in \mathcal{P} . We assume that all edge formulas in the automaton \mathcal{A}_ϕ , and formulas $\text{init}_{\langle \pi_i \rangle}$ and $(\xi_i)_{\langle \pi_i \rangle}$ for $\pi_i \in \mathcal{V}$ are expressible in \mathcal{P} . Note that we can always add missing predicates to \mathcal{P} .

Given the set of predicates \mathcal{P} , the state-space of the abstraction w.r.t. \mathcal{P} is given by \mathbb{B}^n , where for each abstract state $\hat{s} \in \mathbb{B}^n$, the i th position $\hat{s}[i] \in \mathbb{B}$ tracks whether or not predicate p_i holds. To simplify notation, we write $\text{ite}(b, \theta, \theta')$ to be formula θ if $b = \top$, and θ' otherwise. For each abstract state $\hat{s} \in \mathbb{B}^n$, we define $\llbracket \hat{s} \rrbracket := \bigwedge_{i=1}^n \text{ite}(\hat{s}[i], p_i, \neg p_i)$, i.e., $\llbracket \hat{s} \rrbracket$ is a formula over \vec{X} that captures all concrete states that are abstracted to \hat{s} . To incorporate reductions in our abstraction, we parametrize the abstract transition relation by a *scheduling* $M \subseteq \{\pi_1, \dots, \pi_k\}$. We lift the *step* formula from \mathcal{T} by defining

$$\text{step}_M := \bigwedge_{i=1}^k \text{ite}\left(\pi_i \in M, \text{step}_{\langle \pi_i \rangle}, \bigwedge_{x \in X} x'_{\pi_i} = x_{\pi_i}\right).$$

That is all copies in M take a step while all other copies remain unchanged. Given two abstract states \hat{s}_1, \hat{s}_2 we say that \hat{s}_2 is an M -successor of \hat{s}_1 , written $\hat{s}_1 \xrightarrow{M} \hat{s}_2$, if $\llbracket \hat{s}_1 \rrbracket \wedge \llbracket \hat{s}_2 \rrbracket^{(')} \wedge \text{step}_M$ is satisfiable, i.e., we can transition from \hat{s}_1 to \hat{s}_2 by only progressing the copies in M .

For an abstract state \hat{s} , we define $\text{obs}(\hat{s}) \in \mathbb{B}^k$ as the boolean vector that indicates which copy (of π_1, \dots, π_k) is currently at an observation point, i.e., $\text{obs}(\hat{s})[i] = \top$ iff $\llbracket \hat{s} \rrbracket \wedge (\xi_i)_{\langle \pi_i \rangle}$ is satisfiable. Note that as $(\xi_i)_{\langle \pi_i \rangle}$ is, by assumption, expressible in \mathcal{P} , either all or none of the concrete states in $\llbracket \hat{s} \rrbracket$ satisfy $(\xi_i)_{\langle \pi_i \rangle}$.

Game Construction. Building on the parametrized abstract transition relation, we can construct a (finite-state) safety game where winning strategies for the verifier correspond to valid reductions with accompanying proofs. The nodes in our game have two forms: Either they are of the form (\hat{s}, q, b) where $\hat{s} \in \mathbb{B}^n$ is an abstract state, $q \in Q_\phi$ a state of the safety automaton, and $b \in \mathbb{B}^k$ a boolean vector indicating which copy has moved since the last automaton step; Or of the form (\hat{s}, q, b, M) where \hat{s}, q , and b are as before and $\emptyset \neq M \subseteq \{\pi_1, \dots, \pi_k\}$ is a scheduling. The initial states are all states $(\hat{s}, q_{\phi,0}, \top^k)$ where $\llbracket \hat{s} \rrbracket \wedge \bigwedge_{i=1}^k \text{init}_{\langle \pi_i \rangle}$ is satisfiable (recall that $\text{init}_{\langle \pi_i \rangle}$ is expressible in \mathcal{P}). We mark a state (\hat{s}, q, b) or (\hat{s}, q, b, M) as losing iff $q \in B_\phi$. For automaton state $q \in Q_\phi$ and abstract state \hat{s} , we define $\delta_\phi(q, \hat{s})$ as the *unique* state q' such that there is an edge $(q, \theta, q') \in \delta_\phi$ such that $\llbracket \hat{s} \rrbracket \wedge \theta$ is satisfiable. Uniqueness follows from the assumption that \mathcal{A}_ϕ is deterministic and all edge formulas are expressible in \mathcal{P} . The transition relation of our game is given by the following rules:

$$\frac{\forall \pi_i \in M. \neg b[i] \vee \neg \text{obs}(\hat{s})[i]}{(\hat{s}, q, b) \rightsquigarrow (\hat{s}, q, b, M)} \quad \mathbf{(1)} \qquad \frac{\text{obs}(\hat{s}) = \top^k \quad q' = \delta_\phi(q, \hat{s})}{(\hat{s}, q, \top^k) \rightsquigarrow (\hat{s}, q', \perp^k)} \quad \mathbf{(2)}$$

$$\frac{\hat{s} \xrightarrow{M} \hat{s}' \quad b' = b[i \mapsto \top]_{\pi_i \in M}}{(\hat{s}, q, b, M) \rightsquigarrow (\hat{s}', q, b')} \quad \mathbf{(3)}$$

In rule **(1)**, we select any scheduling that schedules only copies that have not reached an observation point or have not moved since the last automaton step. In particular, we cannot schedule any copy that has moved and already reached an observation point. In rule **(2)**, all copies reached an observation point and have moved since the last update (i.e., $b = \top^k$) so we progress the automaton and reset b . Lastly, in rule **(3)**, we select an M -successor of \hat{s} and update b for all copies that take part in the step. In our game, player **SAFE** takes the role of the verifier, and player **REACH** that of the refuter. It is the safety player's responsibility to select a scheduling in each step, so we assign nodes of the form (\hat{s}, q, b) to **SAFE**. Nodes of the form (\hat{s}, q, b, M) are controlled by **REACH** who can choose an abstract M -successor. Let $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^\vee$ be the resulting (finite-state) safety game. A winning strategy for **SAFE** in $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^\vee$ picks, in each abstract state, a valid scheduling that prevents a visit to a losing state. We can thus show:

Theorem 2. *If player **SAFE** wins $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^\vee$, then $\mathcal{T} \models \varphi$.*

Proof Sketch. Assume σ is a winning strategy for **SAFE** in $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^\forall$. Let $t_1, \dots, t_k \in \text{Traces}(\mathcal{T})$ be arbitrary. We, iteratively, construct stuttered versions t'_1, \dots, t'_k of t_1, \dots, t_k by querying σ on abstracted prefixes of t_1, \dots, t_k : Whenever σ schedules copy i we take a proper step on t_i ; otherwise we stutter. By construction of $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^\forall$ the stuttered traces t'_1, \dots, t'_k align at observation points. In particular, we have $[\pi_1 \mapsto \langle t_1 \rangle_{\xi_1}, \dots, \pi_k \mapsto \langle t_k \rangle_{\xi_k}] \models \phi$ iff $[\pi_1 \mapsto \langle t'_1 \rangle_{\xi_1}, \dots, \pi_k \mapsto \langle t'_k \rangle_{\xi_k}] \models \phi$. Moreover, the sequence of abstract states in $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^\forall$ forms an abstraction of t'_1, \dots, t'_k and shows that \mathcal{A}_ϕ cannot reach a bad state when reading $\langle t'_1 \rangle_{\xi_1}, \dots, \langle t'_k \rangle_{\xi_k}$ (as σ is winning). This already shows that $[\pi_1 \mapsto \langle t'_1 \rangle_{\xi_1}, \dots, \pi_k \mapsto \langle t'_k \rangle_{\xi_k}] \models \phi$ and thus $[\pi_1 \mapsto \langle t_1 \rangle_{\xi_1}, \dots, \pi_k \mapsto \langle t_k \rangle_{\xi_k}] \models \phi$. As this holds for all traces $t_1, \dots, t_k \in \text{Traces}(\mathcal{T})$, we get $\mathcal{T} \models \varphi$ as required. \square

Game Construction and Complexity. If the background theory is decidable, $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^\forall$ can be constructed effectively using at most $2^{|\mathcal{P}|+1} \cdot 2^k$ queries to an SMT solver. Checking if **SAFE** wins $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^\forall$ can be done with a simple fixpoint computation of the attractor in linear time.

Our game-based method of finding a reduction in a given abstraction is closely related to the notation of a *property-directed self-composition* [39]. The previously only known algorithm for finding such a reduction is based on an optimized enumeration [39], which, in the worst case, requires $\mathcal{O}(2^{|\mathcal{P}|+1} \cdot 2^k)$ many enumerations. Our worst-case complexity thus matches the bounds inferred by [39], but avoids the explicit enumeration of reductions (and the concomitant repeated construction of the abstract state-space) and is, as we believe, conceptually simpler to comprehend. Moreover, our game-based technique is the key stepping stone for extending our method beyond k -safety in Sect. 6.

6 Verification Beyond k -Safety

Building on the game-based interpretation of a reduction, we extend our verification beyond \forall^* properties to support $\forall^*\exists^*$ properties. We accomplish this by *combining* the game-based reading of a reduction (as discussed in the previous section) with a game-based reading of existential quantification. For the remainder of this section, fix an STS $\mathcal{T} = (X, \text{init}, \text{step})$ and let

$$\varphi = \forall \pi_1 : \xi_1 \dots \forall \pi_l : \xi_l. \exists \pi_{l+1} : \xi_{l+1} \dots \exists \pi_k : \xi_k. \phi$$

be the OHyperLTL formula we wish to check, i.e., we universally quantify over l traces followed by an existential quantification over $k - l$ traces. We assume that for every existential quantification $\exists \pi_i : \xi_i$ occurring in φ , *valid*(t, ξ_i) holds for every $t \in \text{Traces}(\mathcal{T})$ (we discuss this later in Remark 1).

6.1 Existential Trace Quantification as a Game

The idea of a game-based verification of $\forall^*\exists^*$ properties is to consider a $\forall^*\exists^*$ -property as a game between verifier and refuter [23]. The refuter controls the l universally quantified traces by moving through l copies of the system (thereby

producing traces π_1, \dots, π_l) and the verifier reacts by, incrementally, moving through $k - l$ copies of the system (thereby producing traces π_{l+1}, \dots, π_k). If the verifier has a strategy that ensures that the resulting traces satisfy ϕ , $\mathcal{T} \models \varphi$ holds. We call such a strategy for the verifier a *witness strategy*.

We combine this game-based reading of existential quantification with our game-based interpretation of a reduction by, additionally, letting the verifier control the scheduling of the system. When played on the *concrete* state-space of \mathcal{T} the game proceeds in three stages as follows: 1) The verifier selects a valid scheduling $M \subseteq \{\pi_1, \dots, \pi_k\}$; 2) The refuter selects successor states for all universally quantified copies by fixing an assignment to $X'_{\pi_1}, \dots, X'_{\pi_l}$ (only moving copies scheduled by M); 3) The verifier reacts by choosing successor states for the existentially quantified copies by fixing an assignment to $X'_{\pi_{l+1}}, \dots, X'_{\pi_k}$ (again, only moving copies scheduled by M). Afterward, the process repeats.

As we work within a fixed abstraction of \mathcal{T} , the verifier can, however, not choose concrete successor states directly but only work in the precision captured by the abstraction. Following the general scheme of abstract games, we, therefore, underapproximate the moves available to the verifier [2]. Formally, we abstract the three-stage game outlined before (which was played at the level of concrete states) to a simpler abstract game (consisting of only two stages). In the first stage, the verifier selects both a scheduling M and a *restriction* on the set of abstract successor states, i.e., a set of abstract states A . In the second stage, the refuter cannot choose any abstract successor state (any M -successor in the terminology from Sect. 5), but only successors contained in the restriction A . To guarantee the soundness of this approach, we ensure that the verifier can only pick restrictions that are *valid*, i.e., restrictions that underapproximate the possibilities of the verifier on the level of concrete states.

Game Construction. We modify our game from Sect. 5 as follows. States are either of the form (\hat{s}, q, b) (as in Sect. 5) or of the form (\hat{s}, q, b, M, A) where \hat{s} , q , b , and M are as in Sect. 5, and $A \subseteq \mathbb{B}^n$ is a subset of abstract states (the restriction). To reflect the restriction, we modify transition rules (1) and (3). Rule (2) remains unchanged.

$$\frac{\forall \pi_i \in M. \neg b[i] \vee \neg \text{obs}(\hat{s})[i] \quad \text{validRes}_A^{\hat{s}, M}}{(\hat{s}, q, b) \rightsquigarrow (\hat{s}, q, b, M, A)} \quad \text{(1)} \quad \frac{\hat{s}' \in A \quad b' = b[i \mapsto \top]_{i \in M}}{(\hat{s}, q, b, M, A) \rightsquigarrow (\hat{s}', q, b')} \quad \text{(3)}$$

In rule (1), the safety player (who, again, takes the role of the verifier) selects both a scheduling M and a restriction A such that $\text{validRes}_A^{\hat{s}, M}$ holds (which we define later). The reachability player (who takes the role of the refuter) can, in rule (3), select any successor contained in A .

Valid Restriction. The above game construction depends on the definition of $\text{validRes}_A^{\hat{s}, M}$. Intuitively, A is a valid restriction if it underapproximates the possibilities of a witness strategy that can pick concrete successor states for all existentially quantified traces. That is, for every concrete state in \hat{s} , a witness strategy (on the level of concrete states) can guarantee a move to a concrete state that is abstracted to an abstract state within A . Formally we define $\text{validRes}_A^{\hat{s}, M}$ as follows:

$$\begin{aligned} & \forall \{X_{\pi_i}\}_{i=1}^k. \forall \{X'_{\pi_i}\}_{i=1}^l. \llbracket \hat{s} \rrbracket \wedge \bigwedge_{i=1}^l \text{ite} \left(\pi_i \in M, \text{step}_{\langle \pi_i \rangle}, \bigwedge_{x \in X} x'_{\pi_i} = x_{\pi_i} \right) \\ & \Rightarrow \exists \{X'_{\pi_i}\}_{i=l+1}^k. \bigwedge_{i=l+1}^k \text{ite} \left(\pi_i \in M, \text{step}_{\langle \pi_i \rangle}, \bigwedge_{x \in X} x'_{\pi_i} = x_{\pi_i} \right) \wedge \bigvee_{\hat{s}' \in A} \llbracket \hat{s}' \rrbracket \langle \rangle \end{aligned}$$

It expresses that for all concrete states in $\llbracket \hat{s} \rrbracket$ (assignments to $\{X_{\pi_i}\}_{i=1}^k$) and for all concrete successor states for the universally quantified copies (assignments to $\{X'_{\pi_i}\}_{i=1}^l$), there exist successor states for the existentially quantified copies ($\{X'_{\pi_i}\}_{i=l+1}^k$) such that one of the abstract states in A is reached.

Example 1. With this definition at hand, we can validate the restrictions chosen by the strategy in Fig. 3c. For example, in state α_7 the strategy schedules $M = \{2\}$ and restricts the successor states to $\{\alpha_8\}$ even though abstract state $[(6, 4), a_1 = a_2, x_1 \neq x_2]$ is also a $\{2\}$ -successor of α_7 . If we spell out $\text{validRes}_{\{\alpha_8\}}^{\alpha_7, \{2\}}$ we get

$$\forall X_1 \cup X_2 \cup X'_1. \underbrace{a_1 = a_2}_{\llbracket \alpha_7 \rrbracket} \wedge \left(\bigwedge_{z \in X} z'_1 = z_1 \right) \Rightarrow \exists X'_2. \underbrace{a'_2 = a_2 \wedge y'_2 = y_2}_{\text{step}_{\langle 2 \rangle}} \wedge \underbrace{(a'_1 = a'_2 \wedge x'_1 = x_2)}_{\llbracket \alpha_8 \rrbracket \langle \rangle}$$

where $X = \{a, x, y\}$. Here we assume that $\text{step} := (a' = a \wedge y' = y)$ is the update performed on instruction $x \leftarrow \star_{\mathbb{N}}$ from Q2:3 to Q2:4. The above formula is valid.

Correctness. Call the resulting game $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall \exists}$. The game combines the search for a reduction with that of a witness strategy (both within the precision captured by \mathcal{P}).³ We can show:

Theorem 3. *If player SAFE wins $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall \exists}$, then $\mathcal{T} \models \varphi$.*

Proof Sketch. Let σ be a winning strategy for SAFE in $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall \exists}$. Let $t_1, \dots, t_l \in \text{Traces}(\mathcal{T})$ be arbitrary. We use σ to incrementally construct witness traces t_{l+1}, \dots, t_k by querying σ . In every abstract state \hat{s} , σ selects a scheduling M and a restriction A such that $\text{validRes}_{A}^{\hat{s}, M}$ holds. We plug the current concrete state (reached in our construction of t_{l+1}, \dots, t_k) into the universal quantification of $\text{validRes}_{A}^{\hat{s}, M}$ and get (concrete) witnesses for the existential quantification that, by definition of $\text{validRes}_{A}^{\hat{s}, M}$, are valid successors for the existentially quantified copies in \mathcal{T} . \square

Remark 1. Recall that we assume that for every existential quantification $\exists \pi_i : \xi_i$ occurring in φ and all $t \in \text{Traces}(\mathcal{T})$, $\text{valid}(t, \xi_i)$ holds. This is important to ensure that the safety player (the verifier) cannot avoid observation points forever. We could drop this assumption by strengthening the winning condition in $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall \exists}$ and explicitly state that, in order to win, SAFE needs to visit observations points on existentially quantified traces infinitely many times.

³ In particular, $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall \exists}$ (strictly) generalizes the construction of $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall}$ from Sect. 5: If $k = l$ (i.e., the property is a \forall^* -property) the unique minimal valid restriction from \hat{s} , M is $\{\hat{s}' \mid \hat{s} \xrightarrow{M} \hat{s}'\}$, i.e., the set of all M -successors of \hat{s} . The safety player can thus not be more restrictive than allowing all M -successors (as in $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall}$).

Clairvoyance vs. Abstraction. The cooperation between reduction (the ability of the verifier to select schedulings) and witness strategy (the ability to select restrictions on the successor) can be seen as a limited form of prophecy [1, 14]. By first scheduling the universal copies, the witness strategy can peek at future moves before committing to a successor state, as we e.g., saw in Fig. 3. The “theoretically optimal” reduction is thus a sequential one that first schedules only the universally quantified traces (until an observation point is reached) and thereby provides maximal information for the witness strategy. However, in the context of a fixed abstraction, this reduction is not always optimal. For example, in Fig. 3 the strategy schedules the loop in lock-step which is crucial for generating a proof with simple (linear) invariants. In particular, Fig. 3 does not admit a witness strategy in the lock-step reduction and does not admit a proof with linear invariants in a sequential reduction. Our verification framework, therefore, strikes a delicate balance between clairvoyance needed by the witness strategy and precision captured in the abstraction, further emphasizing why the searches for reduction and witness strategy need to be mutually dependent.

6.2 Constructing and Solving $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$

Constructing the game graph of $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$ requires the identification of all valid restrictions (of which there are exponentially many in the number of abstract states and thus double exponentially many in the number of predicates) each of which requires to solve a quantified SMT query. We propose a more effective algorithm that solves $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$ without constructing it explicitly. Instead, we iteratively refine an abstraction $\tilde{\mathcal{G}}$ of $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$. Our method hinges on the following easy observation:

Algorithm 1. Iterative solver for $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$.

```

1: Input:  $\mathcal{T}, \varphi, \mathcal{P}$ 
2:  $\tilde{\mathcal{G}} := \text{initialApproximation}(\mathcal{T}, \varphi, \mathcal{P})$ 
3: repeat
4:   match  $\text{Solve}(\tilde{\mathcal{G}})$  with
5:     case REACH( $\sigma$ ): return REACH
6:     case SAFE( $\sigma$ ):
7:       for all  $(\hat{s}, M, A) \in \text{Restrictions}(\sigma)$  do
8:         if  $\neg \text{validRes}_{\hat{s}, M}^A$  then
9:           for all  $A' \subseteq A$  do
10:             $\tilde{\mathcal{G}} := \text{Remove}(\tilde{\mathcal{G}}, (\hat{s}, M, A'))$ 
11:           goto 4
12:       return SAFE

```

Lemma 1. For any \hat{s} and M , $\{A \mid \text{validRes}_{\hat{s}, M}^A\}$ is upwards closed (w.r.t. \subseteq).

Our initial abstraction consists of all possible restrictions (even those that might be invalid), i.e., we allow all restrictions of the form (\hat{s}, M, A) where $A \subseteq \{\hat{s}' \mid \hat{s} \xrightarrow{M} \hat{s}'\}$.⁴ This overapproximates the power of the safety player, i.e., a winning strategy for SAFE in $\tilde{\mathcal{G}}$ may not be valid in $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$. To remedy this, we propose the following inner refinement loop: If we find a winning strategy σ for

⁴ Note that $\{\hat{s}' \mid \hat{s} \xrightarrow{M} \hat{s}'\}$ is always a valid restriction. Importantly, we can compute $\{\hat{s}' \mid \hat{s} \xrightarrow{M} \hat{s}'\}$ locally, i.e., by iterating over abstract states opposed to sets of abstract states.

SAFE in $\tilde{\mathcal{G}}$ we check if all restrictions chosen by σ are valid. If this is the case, σ is also winning for $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$ and we can apply Theorem 3. If we find an invalid restriction (\hat{s}, M, A) used by σ , we refine $\tilde{\mathcal{G}}$ by removing not only the restriction (\hat{s}, M, A) but *all* (\hat{s}, M, A') with $A' \subseteq A$ (which is justified by Lemma 1). The algorithm is sketched in Algorithm 1. The subroutine *Restrictions*(σ) returns all restrictions used by σ , i.e., all tuples (\hat{s}, M, A) such that σ uses an edge $(\hat{s}, q, b) \rightsquigarrow (\hat{s}, q, b, M, A)$ for some q, b . *Remove*($\tilde{\mathcal{G}}, (\hat{s}, M, A')$) removes from $\tilde{\mathcal{G}}$ all edges of the form $(\hat{s}, q, b) \rightsquigarrow (\hat{s}, q, b, M, A')$ for some q, b , and *Solve* solves a finite-state safety game. To improve the algorithm further, in line 4 we always compute a maximal safety strategy, i.e., a strategy that selects maximal restrictions (w.r.t. \subseteq) and therefore allows us to eliminate many invalid restrictions from $\tilde{\mathcal{G}}$ simultaneously. For safety games, there always exists such a maximal winning strategy (see e.g. [11]). Note that while $\tilde{\mathcal{G}}$ is large, solving this finite-state game can be done very efficiently. The running time of solving $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$ is dominated by the SMT queries of which our refinement loop, in practice, requires very few.

7 Implementation and Evaluation

When combining Theorem 3 and our iterative solver from Sect. 6.2 we obtain an algorithm to verify $\forall^*\exists^*$ -safety properties within a given abstraction. We have implemented a prototype of our method in a tool we call HyPA. We use Z3 [36] to discharge SMT queries. The input of our tool is provided as an arbitrary STS in the SMTLIB format [5], making it *language independent*. In our programs, we make the program counter explicit, allowing us to track predicates locally [32].

Evaluation for k -Safety. As a special case of $\forall^*\exists^*$ properties, HyPA is also applicable to k -safety verification. We collected an exemplifying suite of programs and k -safety properties from the literature [27, 39–41] and manually translated them into STS (this can be automated easily). The results are given in Table 1. As done by Shemer et al. [39], we already provide a

Table 1. Evaluation of HyPA on k -safety instances. We give the size of the abstract game-space (Size), the time taken to compute the abstraction (t_{abs}), and the overall time taken by HyPA (t). Times are given in seconds.

Instance	Size	t_{abs}	t
DoubleSquareNI	819	92.3	92.8
HalfSquareNI	1166	85.9	86.5
SquaresSum	286	29.8	29.9
ArrayInsert	213	28.2	28.2
Exp1x3	112	4.5	4.5
Fig3	268	11.9	12.0
DoubleSquareNIff	121	9.8	9.9
Fig. 2	333	23.7	23.8
ColItem-Symm	494	24.0	24.1
Counter-Det	216	10.2	10.3
MultEquiv	757	18.9	19.0

set of predicates that is sufficient for *some* reduction (but not necessarily the lockstep or sequential one), the search for which is then automated by HyPA. Our results show the game-based search for a reduction can verify interesting

Table 2. Evaluation of HyPA on $\forall^*\exists^*$ -safety verification instances. We give the size and construction time of the initial abstraction (Size and t_{abs}). For both the direct (explicit) and lazy (Algorithm 1) solver we give the time to construct (and solve) the game (t_{solve}) and the overall time ($t = t_{abs} + t_{solve}$). For the lazy solver we, additionally, give the number of refinement iterations (#Ref). Times are given in seconds. TO indicates a timeout after 5 min.

Instance	Size	t_{abs}	Direct		Lazy		
			t_{solve}	t	#Ref	t_{solve}	t
NonDetAdd	4568	3.5	TO	TO	4	1.0	4.5
CounterSum	479	5.3	9.1	14.4	17	0.9	6.2
AsynchGNI	437	6.1	6.9	13.0	1	0.1	6.2
CompilerOpt1	354	2.4	2.3	4.7	2	0.2	2.6
CompilerOpt2	338	2.8	2.4	5.2	2	0.2	3.0
Refine	1357	6.1	TO	TO	4	0.7	6.8
Refine2	1476	5.6	TO	TO	5	0.6	6.2
Smaller	327	2.3	4.0	6.3	11	0.4	2.7
CounterDiff	959	8.5	18.3	26.8	19	1.1	9.6
Fig. 3	3180	11.1	TO	TO	22	2.9	14.0
P1 (simple)	83	2.0	1.4	3.4	1	0.1	2.1
P1 (GNI)	34793	17.0	TO	TO	72	95.7	112.7
P2 (GNI)	15753	10.2	TO	TO	7	5.1	15.3
P3 (GNI)	1429	6.6	20.9	27.5	7	0.6	7.2
P4 (GNI)	7505	16.5	TO	TO	72	13.2	29.7

k -safety properties from the literature. We also note that, currently, the vast majority of time is spent on the construction of the abstract system. If we would move to a fixed language, the computation time of the initial abstraction could be reduced by using existing (heavily optimized) abstraction tools [18,32].

Evaluation Beyond k -Safety. The main novelty of HyPA lies in its ability to, for the first time, verify temporal properties beyond k -safety. As none of the existing tools can verify such properties, we compiled a collection of very small example programs and $\forall^*\exists^*$ -safety properties. Additionally, we modified the boolean programs from [13] (where they checked GNI on boolean programs) by including data from infinite domains. The properties we checked range from refinement properties for compiler optimizations, over general refinement of non-deterministic programs, to generalized non-interference. Verification often requires a non-trivial combination of reduction and witness strategy (as the reduction must, e.g., compensate for branches of different lengths). As before, we provide

a set of predicates and let HyPA automatically search for a witness strategy with accompanying reduction. We list the results in Table 2. To highlight the effectiveness of our inner refinement loop, we apply both a direct (explicit) construction of $\mathcal{G}_{(\mathcal{T}, \varphi, \mathcal{P})}^{\forall\exists}$ and the lazy (iterative) solver in Algorithm 1. Our lazy solver (Algorithm 1) clearly outperforms an explicit construction and is often the only method to solve the game in reasonable time. In particular, we require very few refinement iterations and therefore also few expensive SMT queries. Unsurprisingly, the problem of verifying properties beyond k -safety becomes much more challenging (compared to k -safety verification) as it involves the *synthesis* of a witness function which is already 2EXPTIME-hard for finite-state systems [23, 37]. We emphasize that no other existing tool can verify any of the benchmarks.

8 Related Work

Asynchronous Hyperproperties. Recently, many logics for the formal specification of asynchronous hyperproperties have been developed [9, 13, 17, 31]. Our logic OHyperLTL is closely related to stuttering HyperLTL (HyperLTL_S) [17]. In HyperLTL_S each temporal operator is endowed with a set of temporal formulas Γ and steps where the truth values of all formulas in Γ remain unchanged are ignored during the operator’s evaluation. As for most mechanisms used to design asynchronous hyperlogics [9, 17, 31], finite-state model checking of HyperLTL_S is undecidable. By contrast, in OHyperLTL, we always observe the trace at a fixed location, which is key for ensuring decidable finite-state model checking.

k-Safety Verification. The literature on k -safety verification is rich. Many approaches verify k -safety by using a form of self-composition [8, 20, 25, 28] and often employ reductions to obtain compositions that are easier to verify. Our game-based interpretation of a reduction (Sect. 5) is related to Shemer et al. [39], who study k -safety verification within a given predicate abstraction using an enumeration-based solver (see Sect. 5 for a discussion). Farzan and Vandikas [27] present a counterexample-guided refinement loop that simultaneously searches for a reduction and a proof. Sousa and Dillig [40] facilitate reductions at the source-code level in program logic.

\forall^\exists^* -Verification.* Barthe et al. [7] describe an asymmetric product of the system such that only a subset of the behavior of the second system is preserved, thereby allowing the verification of $\forall^*\exists^*$ properties. Constructing an asymmetric product and verifying its correctness (i.e., showing that the product preserves all behavior of the first, universally quantified, system) is challenging. Unno et al. [41] present a constraint-based approach to verify functional (opposed to temporal) $\forall\exists$ properties in infinite-state systems using an extension of constraint Horn clauses called pfwCHC. The underlying verification approach is orthogonal to ours: pfwCHC allows for a clean separation of the actual verification and verification conditions, whereas our approach combines both. For example, our method can prove the existence of a witness strategy without ever formulating precise constraints on the strategy (which seems challenging). Coenen et

al. [23] introduce the game-based reading of existential quantification to verify temporal $\forall^*\exists^*$ properties in a synchronous and finite-state setting. By contrast, our work constitutes the first verification method for temporal $\forall^*\exists^*$ -safety properties in *infinite-state* systems. The key to our method is a careful integration of reductions which is not possible in a synchronous setting. For finite-state systems (where the abstraction is precise) and synchronous specifications (where we observe every step), our method subsumes the one in [23]. Beutner and Finkbeiner [14] use prophecy variables to ensure that the game-based reading of existential quantification is complete in a finite-state setting. Automatically constructing prophecies for infinite-state systems is interesting future work. Pommellet and Touili [38] study the verification of HyperLTL in infinite-state systems arising from pushdown systems. By contrast, we study verification in infinite-state systems that arise from the infinite variables domains used in software.

Game Solving. Our game-based interpretations are naturally related to infinite-state game solving [4, 16, 26, 42]. State-of-the-art solvers for infinite-state games unroll the game [26], use necessary subgoals to inductively split a game into subgames [4], encode the game as a constraint system [16], and iteratively refine the controllable predecessor operator [42]. We tried to encode our verification approach directly as an infinite-state linear-arithmetic game. However, existing solvers (which, notably, work *without* a user-provided set of predicates) could not solve the resulting game [4, 26]. Our method for encoding the witness strategy using *restrictions* corresponds to hyper-must edges in general abstract games [2, 3]. Our inner refinement loop for solving a game with hyper-must edges without explicitly identifying all edges (Algorithm 1) is thus also applicable in general abstract games.

9 Conclusion

In this work, we have presented the first verification method for temporal hyperproperties beyond k -safety in infinite-state systems arising in software. Our method is based on a game-based interpretation of reductions and existential quantification and allows for mutual dependence of both. Interesting future directions include the integration of our method in a counter-example guided refinement loop that automatically refines the abstraction and ways to lift the current restriction to temporally safe specifications. Moreover, it is interesting to study if, and to what extent, the numerous other methods developed for k -safety verification of infinite-state systems (apart from reductions) are applicable to the vast landscape of hyperproperties that lies beyond k -safety.

Acknowledgments. This work was partially supported by the DFG in project 389792660 (Center for Perspicuous Systems, TRR 248). R. Beutner carried out this work as a member of the Saarbrücken Graduate School of Computer Science.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. *Theor. Comput. Sci.* **82**(2), 253–284 (1991). [https://doi.org/10.1016/0304-3975\(91\)90224-P](https://doi.org/10.1016/0304-3975(91)90224-P)
2. de Alfaro, L., Godefroid, P., Jagadeesan, R.: Three-valued abstractions of games: uncertainty, but with precision. In: *IEEE Symposium on Logic in Computer Science, LICS 2004*. IEEE (2004). <https://doi.org/10.1109/LICS.2004.1319611>
3. de Alfaro, L., Roy, P.: Solving games via three-valued abstraction refinement. In: *Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007*. LNCS, vol. 4703, pp. 74–89. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_6
4. Baier, C., Coenen, N., Finkbeiner, B., Funke, F., Jantsch, S., Siber, J.: Causality-based game solving. In: *Silva, A., Leino, K.R.M. (eds.) CAV 2021*. LNCS, vol. 12759, pp. 894–917. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_42
5. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: *International Workshop on Satisfiability Modulo Theories*, vol. 13 (2010)
6. Barrett, C., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.: TVOC: a translation validator for optimizing compilers. In: *Etessami, K., Rajamani, S.K. (eds.) CAV 2005*. LNCS, vol. 3576, pp. 291–295. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_29
7. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: asymmetric product programs for relational program verification. In: *Artemov, S., Nerode, A. (eds.) LFCS 2013*. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35722-0_3
8. Barthe, G., D’Argenio, P.R., Rezk, T.: Secure information flow by self-composition. *Math. Struct. Comput. Sci.* **21**(6), 1207–1252 (2011). <https://doi.org/10.1017/S0960129511000193>
9. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: *Silva, A., Leino, K.R.M. (eds.) CAV 2021*. LNCS, vol. 12759, pp. 694–717. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_33
10. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: *ACM Symposium on Principles of Programming Languages, POPL 2004*. ACM (2004). <https://doi.org/10.1145/964001.964003>
11. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: from parity games to safety games. *RAIRO Theor. Inf. Appl.* **36**(3), 261–275 (2002). <https://doi.org/10.1051/ita:2002013>
12. Beutner, R., Carral, D., Finkbeiner, B., Hofmann, J., Krötzsch, M.: Deciding hyperproperties combined with functional specifications. In: *Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2022*. ACM (2022). <https://doi.org/10.1145/3531130.3533369>
13. Beutner, R., Finkbeiner, B.: A temporal logic for strategic hyperproperties. In: *International Conference on Concurrency Theory, CONCUR 2021*. LIPIcs, vol. 203. Schloss Dagstuhl (2021). <https://doi.org/10.4230/LIPIcs.CONCUR.2021.24>
14. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: *IEEE Computer Security Foundations Symposium, CSF 2022*. IEEE (2022)
15. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k -safety. *CoRR* (2022). <https://doi.org/10.48550/arXiv.2206.03381>

16. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: Annual ACM Symposium on Principles of Programming Languages, POPL 2014. ACM (2014). <https://doi.org/10.1145/2535838.2535860>
17. Bozzelli, L., Peron, A., Sánchez, C.: Asynchronous extensions of HyperLTL. In: Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021. IEEE (2021). <https://doi.org/10.1109/LICS52264.2021.9470583>
18. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. *IEEE Trans. Softw. Eng.* **30**(6), 388–402 (2004). <https://doi.org/10.1109/TSE.2004.22>
19. Chaudhuri, S., Gulwani, S., Lubliner, R.: Continuity and robustness of programs. *Commun. ACM* **55**(8), 107–115 (2012). <https://doi.org/10.1145/2240236.2240262>
20. Churchill, B.R., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019. ACM (2019). <https://doi.org/10.1145/3314221.3314596>
21. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
22. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: IEEE Computer Security Foundations Symposium, CSF 2008. IEEE (2008). <https://doi.org/10.1109/CSF.2008.7>
23. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 121–139. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_7
24. D’Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 47–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_3
25. Eilers, M., Müller, P., Hitz, S.: Modular product programs. *ACM Trans. Program. Lang. Syst.* **42**(1), 1–37 (2020). <https://doi.org/10.1145/3324783>
26. Farzan, A., Kincaid, Z.: Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang.* **2**(POPL), 1–30 (2018). <https://doi.org/10.1145/3158149>
27. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 200–218. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_11
28. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
29. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* **8**(1), 1–27 (2016). <https://doi.org/10.1007/s13389-016-0141-6>
30. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
31. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.* **5**(POPL), 1–29 (2021). <https://doi.org/10.1145/3434319>

32. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: ACM Symposium on Principles of Programming Languages, POPL 2002. ACM (2002). <https://doi.org/10.1145/503272.503279>
33. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_15
34. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975). <https://doi.org/10.1145/361227.361234>
35. McCullough, D.: Noninterference and the composability of security properties. In: IEEE Symposium on Security and Privacy, SP 1988. IEEE (1988). <https://doi.org/10.1109/SECPRI.1988.8110>
36. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
37. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Annual ACM Symposium on Principles of Programming Languages, POPL 1989. ACM (1989). <https://doi.org/10.1145/75277.75293>
38. Pommellet, A., Touili, T.: Model-checking HyperLTL for pushdown systems. In: Gallardo, M.M., Merino, P. (eds.) SPIN 2018. LNCS, vol. 10869, pp. 133–152. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94111-0_8
39. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 161–179. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_9
40. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016. ACM (2016). <https://doi.org/10.1145/2908080.2908092>
41. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 742–766. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_35
42. Walker, A., Ryzhyk, L.: Predicate abstraction for reactive synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2014. IEEE (2014). <https://doi.org/10.1109/FMCAD.2014.6987617>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

