Acta Informatica (2020) 57:137–163
https://doi.org/10.1007/s00236-019-00358-2

ORIGINAL ARTICLE

Synthesis from hyperproperties

Bernd Finkbeiner[1] · Christopher Hahn[1] · Philip Lukert[1] · Marvin Stenger[1] · Leander Tentrup[1]

Received: 15 January 2019 / Accepted: 22 November 2019 / Published online: 7 December 2019
© The Author(s) 2019

Abstract
We study the reactive synthesis problem for hyperproperties given as formulas of the temporal logic HyperLTL. Hyperproperties generalize trace properties, i.e., sets of traces, to *sets of sets* of traces. Typical examples are information-flow policies like noninterference, which stipulate that no sensitive data must leak into the public domain. Such properties cannot be expressed in standard linear or branching-time temporal logics like LTL, CTL, or CTL*. Furthermore, HyperLTL subsumes many classical extensions of the LTL realizability problem, including realizability under incomplete information, distributed synthesis, and fault-tolerant synthesis. We show that, while the synthesis problem is undecidable for full HyperLTL, it remains decidable for the $\exists^*$, $\exists^*\forall^1$, and the *linear* $\forall^*$ fragments. Beyond these fragments, the synthesis problem immediately becomes undecidable. For universal HyperLTL, we present a semi-decision procedure that constructs implementations and counterexamples up to a given bound. We report encouraging experimental results obtained with a prototype implementation on example specifications with hyperproperties like symmetric responses, secrecy, and information flow.

1 Introduction

Hyperproperties [9] generalize trace properties in that they not only check the correctness of *individual* computation traces in isolation, but relate *multiple* computation traces to each other. HyperLTL [8] is a logic for expressing temporal hyperproperties, by extending linear-time temporal logic (LTL) with *explicit* quantification over traces. HyperLTL has been used to specify a variety of information-flow and security properties. Examples include classical security properties like non-interference [32] and observational determinism [38], as well as quantitative information-flow properties [7], symmetries in hardware designs, and formally verified error correcting codes [26]. For example, observational determinism can be expressed as the HyperLTL formula $\forall\pi\forall\pi'.\square(I_\pi = I_{\pi'}) \rightarrow \square(O_\pi = O_{\pi'})$, stating that, for every pair

This work was partially supported by the Collaborative Research Centers "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223) and "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

✉ Leander Tentrup
tentrup@react.uni-saarland.de

1 Reactive Systems Group, Saarland University, Saarbrücken, Germany

of traces, if the observable inputs are the same, then the observable outputs must be same as well.

In reactive synthesis, we automatically construct an implementation that is guaranteed to satisfy a given specification. A fundamental difference to verification is that there is no human programmer involved: in verification, the programmer would first produce an implementation, which is then verified against the specification. In synthesis, the implementation is directly constructed from the specification. Because there is no programmer, it is crucial that the specification contains *all* desired properties of the implementation: the synthesized implementation is guaranteed to satisfy the given specification, but nothing is guaranteed beyond that. The added expressive power of HyperLTL over LTL is very attractive for synthesis: with synthesis from hyperproperties, we can guarantee that the implementation does not only accomplish the desired functionality, but is also free of information leaks, is symmetric, is fault-tolerant with respect to transmission errors, etc.

More formally, the reactive synthesis problem asks for a *strategy*, that is a tree which branches on environment inputs and whose nodes are labeled by the system output. Collecting the inputs and outputs along a branch of the tree, we obtain a trace. If the set of traces collected from the branches of the strategy tree satisfies the specification, we say that the strategy *realizes* the specification. The specification is *realizable* iff there exists a strategy tree that realizes the specification. With LTL specifications, we get trees where the trace on each individual branch satisfies the LTL formula. With HyperLTL, we get trees where the traces between different branches are in a specified relationship. This is dramatically more powerful.

Consider, for example, the well-studied *distributed* version of the reactive synthesis problem, where the system is split into a set of processes, where each only sees a subset of the inputs. The distributed synthesis problem for LTL can be expressed as the standard (non-distributed) synthesis problem for HyperLTL, by adding for each process the requirement that the process output is *observationally deterministic* in the process input. HyperLTL synthesis thus subsumes distributed synthesis. The information-flow requirements realized by HyperLTL synthesis can, however, be much more sophisticated than the observational determinism needed for distributed synthesis. Consider, for example, the *dining cryptographers* problem [6]: three cryptographers C_a, C_b, and C_c sit at a table in a restaurant having dinner and either one of the cryptographers or, alternatively, the NSA must pay for their meal. Is there a protocol where each cryptographer can find out whether it was a cryptographer who paid or the NSA, but cannot find out which cryptographer paid the bill?

Synthesis from LTL formulas is known to be decidable in doubly exponential time [39]. The fact that the distributed synthesis problem is undecidable [40] immediately eliminates the hope for a similar general result for HyperLTL. However, since LTL is obviously a fragment of HyperLTL, this immediately leads to the question whether the synthesis problem is still decidable for fragments of HyperLTL that are close to LTL but go beyond LTL: when exactly does the synthesis problem become undecidable? From a more practical point of view, the interesting question is whether semi-algorithms for distributed synthesis [16,28], which have been successful in constructing distributed systems from LTL specifications despite the undecidability of the general problem, can be extended to HyperLTL?

In this paper, we answer the first question by studying the $\exists^*$, $\exists^*\forall^1$, and the *linear* $\forall^*$ fragment. We show that the synthesis problem for all three fragments is decidable, and the problem becomes undecidable as soon as we go beyond these fragments. In particular, the synthesis problem for the full $\forall^*$ fragment, which includes observational determinism, is undecidable.

We answer the second question by studying the *bounded* version of the synthesis problem for the $\forall^*$ fragment. In order to detect realizability, we ask whether, for a universal HyperLTL formula φ and a given bound n on the number of states, there exists a representation of the strategy tree as a finite-state machine with no more than n states that satisfies φ. To detect unrealizability, we check whether there exists a counterexample to realizability of bounded size. We show that both checks can be effectively reduced to SMT solving.

1.1 Related work

HyperLTL [8] is a successor of the temporal logic SecLTL [14] used to characterize temporal information flow. The model-checking [8,25,26], satisfiability [18,19,21], monitoring problem [1–4,22–24,33,34], and the first-order extension [31] of HyperLTL have been studied before. In [11], it has been shown that existential quantification in a HyperLTL formula can be reduced to strategic choice. An extensive study of the hierarchy of hyperlogics beyond HyperLTL has been initiated in [10].

We base our algorithms on well-known synthesis algorithms such as bounded synthesis [28] that itself is an instance of Safraless synthesis [36] for ω-regular languages. A further technique that we adapt for hyperproperties is the bounded unrealizability method [29,30].

Hyperproperties [9] can be seen as a unifying framework for many different properties of interest in multiple distinct areas of research. Information-flow properties in security and privacy research are hyperproperties [8]. HyperLTL subsumes logics that reason over knowledge [8]. Information flow in distributed systems is another example of hyperproperties, and the HyperLTL realizability problem subsumes both the distributed synthesis problem [27,40] as well as synthesis of fault-tolerant systems [30]. In circuit verification, the semantic independence of circuit output signals on a certain set of inputs, enabling a range of potential optimizations, is a hyperproperty.

This article is an extended version of [20], including previously omitted proofs. Additionally, we show that HyperLTL realizability extends many previous extensions to LTL realizability, including realizability under incomplete information [35], distributed synthesis [27,40], and fault-tolerant synthesis [30].

1.2 Structure of this article

We introduce HyperLTL and necessary preliminaries in Sect. 2. In Sect. 3 we define the realizability problem for HyperLTL and demonstrate the expressiveness compared to classical extensions of LTL realizability. In the following section, we investigate the decidability for the realizability problem, where we characterize fragments based on the quantifier prefix. In Sects. 5 and 6 we give algorithms for the bounded realizability and unrealizability problem of universal HyperLTL, i.e., we bound the size of the system and environment, respectively, in order to derive a semi-decision procedure. We report on experimental evaluation of our prototype synthesis tool on a variety of benchmarks, involving distributed architectures, fault-tolerance, and secrecy properties.

2 Preliminaries

2.1 HyperLTL

HyperLTL [8] is a temporal logic for specifying hyperproperties. It extends LTL by quantification over trace variables π and a method to link atomic propositions to specific traces. The set of trace variables is $\mathcal{V}$. Formulas in HyperLTL are given by the grammar

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

where $a \in \mathrm{AP}$ and $\pi \in \mathcal{V}$. The alphabet of a HyperLTL formula is 2^{AP}. We allow the standard boolean connectives $\wedge$, $\rightarrow$, $\leftrightarrow$ as well as the derived LTL operators release $\varphi\, \mathcal{R}\, \psi \equiv \neg(\neg\varphi\, \mathcal{U} \neg\psi)$, eventually $\Diamond \varphi \equiv \mathit{true}\, \mathcal{U}\, \varphi$, globally $\Box \varphi \equiv \neg \Diamond \neg \varphi$, and weak until $\varphi\, \mathcal{W}\, \psi \equiv \Box \varphi \vee (\varphi\, \mathcal{U}\, \psi)$.

The semantics is given by the satisfaction relation $\vDash_T$ over a set of traces $T \subseteq (2^{\mathrm{AP}})^\omega$. We define an assignment $\Pi : \mathcal{V} \rightarrow (2^{\mathrm{AP}})^\omega$ that maps trace variables to traces. $\Pi[i, \infty]$ is the trace assignment that for every π maps Π to the trace $\Pi(\pi)[i, \infty]$, i.e., it removes the first i items from the traces.

$\Pi \vDash_T a_\pi$	if $a \in \Pi(\pi)[0]$
$\Pi \vDash_T \neg\varphi$	if $\Pi \nvDash_T \varphi$
$\Pi \vDash_T \varphi \vee \psi$	if $\Pi \vDash_T \varphi$ or $\Pi \vDash_T \psi$
$\Pi \vDash_T \bigcirc \varphi$	if $\Pi[1, \infty] \vDash_T \varphi$
$\Pi \vDash_T \varphi\, \mathcal{U}\, \psi$	if $\exists i \geq 0.\, \Pi[i, \infty] \vDash_T \psi \wedge \forall 0 \leq j < i .\, \Pi[j, \infty] \vDash_T \varphi$
$\Pi \vDash_T \exists \pi .\, \varphi$	if there is some $t \in T$ such that $\Pi[\pi \mapsto t] \vDash_T \varphi$
$\Pi \vDash_T \forall \pi .\, \varphi$	if for all $t \in T$ holds that $\Pi[\pi \mapsto t] \vDash_T \varphi$

We write $T \vDash \varphi$ for $\{\} \vDash_T \varphi$ where $\{\}$ denotes the empty assignment. Two HyperLTL formulas φ and ψ are equivalent, written $\varphi \equiv \psi$ if they have the same models. A HyperLTL formula φ is denoted satisfiable if there is a set of traces T which satisfies φ, i.e., $T \vDash \varphi$. The satisfiability problem is undecidable for general HyperLTL formulas but becomes decidable if we renounce $\forall^*\exists^*$ formulas which alternate the quantifier from $\forall$ to $\exists$ [18]. For an LTL formula φ, we denote by $\varphi[\pi]$ the quantifier-free HyperLTL formula where every proposition a is replaced by a_π.

(In)dependence is a hyperproperty that we will use multiple times in this article, thus, we define the following syntactic sugar. Given two disjoint subsets of atomic propositions $C \subseteq \mathrm{AP}$ and $A \subseteq \mathrm{AP}$, we define independence as the following HyperLTL formula

$$D_{A \mapsto C}^{\pi, \pi'} := \left(\bigvee_{a \in A} (a_\pi \nleftrightarrow a_{\pi'}) \right) \mathcal{R} \left(\bigwedge_{c \in C} (c_\pi \leftrightarrow c_{\pi'}) \right), \tag{1}$$

which requires that the valuations of propositions C on traces π and π' have to be equal until and including the point in time where there is a difference in the valuation of some proposition in A. Prefacing universal quantification, that is, the formula $\forall \pi \forall \pi' .\, D_{A \mapsto C}^{\pi, \pi'}$ guarantees that every proposition $c \in C$ solely depends on propositions in A.

2.2 Strategies

A *strategy* $f\colon (2^I)^* \rightarrow 2^O$ maps sequences of input valuations 2^I to an output valuation 2^O. The behavior of a strategy $f\colon (2^I)^* \rightarrow 2^O$ is characterized by an infinite tree, called *computation tree*, that branches by the valuations of I and whose nodes $w \in (2^I)^*$ are labeled with the strategic choice $f(w)$. For an infinite word $w = w_0 w_1 w_2 \cdots \in (2^I)^\omega$, the corresponding *trace* is defined as $(f(\epsilon) \cup w_0)(f(w_0) \cup w_1)(f(w_0 w_1) \cup w_2) \cdots \in (2^{I \cup O})^\omega$. We lift the set containment operator $\in$ to the containment of a trace $w = w_0 w_1 w_2 \cdots \in (2^{I \cup O})^\omega$ in a strategy tree induced by $f\colon (2^I)^* \rightarrow 2^O$, i.e., $w \in f$ if, and only if, $f(\epsilon) = w_0 \cap O$ and $f((w_0 \cap I) \cdots (w_i \cap I)) = w_{i+1} \cap O$ for all $i \geq 0$. The set of traces produced by f, written *traces*(f), is $\{w \mid w \in f\}$. We define the satisfaction of a HyperLTL formula φ (over propositions $I \cup O$) on strategy f, written $f \vDash \varphi$, as *traces*$(f) \vDash \varphi$. Thus, a strategy f is a model of φ if the set of traces of f is a model of φ.

3 HyperLTL synthesis

In this section, we introduce the realizability problem for HyperLTL and compare its expressiveness to various previous extensions of the LTL realizability problem.

Definition 1 (HyperLTL *Realizability*) A HyperLTL formula φ over atomic propositions $\mathrm{AP} = I \mathbin{\dot\cup} O$ is realizable if there is a strategy $f\colon (2^I)^* \rightarrow 2^O$ that satisfies φ.

The fragment of HyperLTL with only a single, universal quantifier $\forall \pi.\, \varphi$ is equivalent to the LTL realizability problem of φ. With two universal quantifiers, one can express relations between traces in the execution tree, thus, one can express the LTL realizability problem with restricted information flow like incomplete information [35], distributed synthesis [27,40], and fault-tolerant synthesis [13,30].

3.1 Incomplete information

The realizability problem with incomplete information [35] is a tuple $\langle \varphi, I, O, H \rangle$, where φ is an LTL formula, I is a set of input propositions, O is a set of output propositions, and $H \subseteq I$ is a set of hidden inputs not observable by the system. Thus, a realizing strategy $f\colon (2^{I \setminus H})^* \rightarrow 2^O$ has a computation tree that only branches by $I \setminus H$. In order to evaluate φ, which may include propositions H, the computation tree is *widened* [35] by H. In HyperLTL, we can verify that a strategy $f'\colon (2^I)^* \rightarrow 2^O$ has the same output-behavior as a H-widened strategy f by checking $f' \vDash \forall \pi \forall \pi'.\, D^{\pi,\pi'}_{I \setminus H \mapsto O}$.

Theorem 1 *The HyperLTL realizability problem subsumes the LTL realizability with incomplete information problem.*

Proof Given $\langle \varphi, I, O, H \rangle$, the following HyperLTL formula over inputs I and outputs O is equirealizable.

$$\forall \pi \forall \pi'.\, \varphi[\pi] \wedge D^{\pi,\pi'}_{I \setminus H \mapsto O}$$

□

(a) An architecture of two processes that specify process p_1 to produce c from a and p_2 to produce d from b.

(b) The same architecture as on the left, where the inputs of process p_2 are changed to a and b.

Fig. 1 Distributed architectures

3.2 Distributed synthesis

The distributed synthesis problem was introduced by Pnueli and Rosner [40] and introduces the concept of *architectures* as a constraint on the information flow. An architecture is a set of processes P, with distinct environment process $p_{env} \in P$, such that the processes produce outputs synchronously, but each process bases its decision only on the history of valuation of inputs that it observes.

Formally, a distributed architecture $\mathcal{A}$ is a tuple $\langle P, p_{env}, \mathcal{I}, \mathcal{O}\rangle$ where P is a finite set of processes with distinguished environment process $p_{env} \in P$. The functions $\mathcal{I}\colon P \to 2^{\text{AP}}$ and $\mathcal{O}\colon P \to 2^{\text{AP}}$ define the inputs and outputs of processes. While processes may share the same inputs (in case of broadcasting), the outputs of processes must be pairwise disjoint, i.e., for all $p \neq p' \in P$ it holds that $\mathcal{O}(p) \cap \mathcal{O}(p') = \emptyset$. W.l.o.g. we assume that $\mathcal{I}(p_{env}) = \emptyset$. We denote by $P^- = P \setminus \{p_{env}\}$ the set of processes excluding the environment process.

The distributed realizability problem for architectures without *information forks* [27] is decidable. Intuitively, an information fork is a situation where two distinct processes $p, p' \in P$ receive environment inputs I and I' (may be transitive through other processes) such that both observe inputs that the other process does not observe, i.e., there exist $i \in I$ and $i' \in I'$ such that $i \notin I'$ and $i' \notin I$. We depict two example architectures in Fig. 1. The architecture in Fig. 1a contains an information fork while the architecture in Fig. 1b does not. Furthermore, the processes in Fig. 1b can be ordered linearly according to the subset relation on the inputs.

Theorem 2 *The HyperLTL realizability problem subsumes the distributed LTL realizability problem.*

Proof Given a distributed realizability problem $\langle \varphi, \mathcal{A}\rangle$, the following HyperLTL formula over inputs $\mathcal{O}(p_{env})$ for P^- and outputs $\bigcup_{p \in P^-} \mathcal{O}(p)$ is equirealizable:

$$\forall \pi \forall \pi'.\, \varphi[\pi] \wedge \bigwedge_{p \in P^-} D^{\pi,\pi'}_{\mathcal{I}(p) \mapsto \mathcal{O}(p)}$$

□

3.3 Asynchronous distributed synthesis

The asynchronous system model [42] is a generalization of the synchronous model discussed previously. In this model, we have a global scheduler, controlled by the environment, that

decides when and which processes are scheduled. The resulting distributed realizability problem is already undecidable for LTL specifications and systems with more than one process [42].

Theorem 3 *The HyperLTL realizability problem subsumes the asynchronous distributed LTL realizability problem.*

Proof Let $\mathcal{A} = \langle P, p_{env}, \mathcal{I}, \mathcal{O} \rangle$ be a distributed architecture. To model scheduling, we introduce an additional set $Sched = \{sched_p \mid p \in P^-\}$ of atomic propositions. The valuation of $sched_p$ indicates whether system process p is currently scheduled or not. A process $p \in P^-$ may observe whether it is scheduled or not, that is, it may depend on $sched_p$. The environment can decide at every step which processes to schedule. When a process is not scheduled, its output behavior does not change [28]. As the scheduling is controlled by the environment, we assume that every process is infinitely often scheduled, as otherwise, the environment wins by simply not scheduling any process.

Given an asynchronous distributed realizability problem $\langle \varphi, \mathcal{A} \rangle$, the following HyperLTL formula over inputs $\mathcal{O}(p_{env}) \cup Sched$ and outputs $\bigcup_{p \in P^-} \mathcal{O}_p$ is equirealizable:

$$\forall \pi \forall \pi'. \left(\bigwedge_{p \in P^-} \square \diamondsuit sched_p[\pi] \right) \rightarrow \varphi[\pi] \wedge \bigwedge_{p \in P^-} D^{\pi,\pi'}_{(\mathcal{I}(p) \cup \{sched_p\}) \mapsto \mathcal{O}(p)}$$
$$\wedge \square \bigwedge_{p \in P^-} \neg sched_p[\pi] \rightarrow \left(\bigwedge_{o \in \mathcal{O}(p)} o_\pi \leftrightarrow \bigcirc o_\pi \right)$$

□

3.4 Symmetric synthesis

A special case of distributed synthesis is symmetric synthesis [15], which, additionally to distributivity, requires that all system processes act exactly the same if they are given the same inputs. Formally, symmetric synthesis requires a symmetric architecture $\langle P, p_{env}, \mathcal{I}, \mathcal{O} \rangle$ where for each process $p \in P^-$, $|\mathcal{I}(p)| = n$ and $|\mathcal{O}(p)| = m$ for some $n, m \in \mathbb{N}$. We assume an implicit ordering of inputs and output per process and use the notation $\mathcal{I}(p)_j$ and $\mathcal{O}(p)_j$ to access the j-th input and output of process $p \in P^-$, respectively. Then, we can express the symmetry constraint as an LTL formula

$$\bigwedge_{p, p' \in P^-} \left(\bigvee_{1 \le j \le n} \mathcal{I}(p)_j \not\leftrightarrow \mathcal{I}(p')_j \right) \mathcal{R} \left(\bigwedge_{1 \le j \le m} \mathcal{O}(p)_j \leftrightarrow \mathcal{O}(p')_j \right) \quad \text{(sym)}$$

Theorem 4 *The HyperLTL realizability problem subsumes the symmetric (distributed) LTL realizability problem.*

Proof Given a symmetric realizability problem over architecture $\mathcal{A}$ and specifications $\varphi_1, \ldots, \varphi_k$ for the k processes the following HyperLTL formula over inputs $\mathcal{O}(p_{env})$ for P^- and outputs $\bigcup_{p \in P^-} \mathcal{O}(p)$ is equirealizable:

$$\forall \pi \forall \pi'. \bigwedge_{1 \le i \le k} \varphi_i[\pi] \wedge \bigwedge_{p \in P^-} D^{\pi,\pi'}_{\mathcal{I}(p) \mapsto \mathcal{O}(p)} \wedge \text{sym}[\pi]$$

□

3.5 Fault-tolerant synthesis

We consider another extension to the distributed synthesis problem where we incorporate the possibility that communication *between* processes may be subject to faults, such as Byzantine faults [29,30]. In the distributed synthesis formulation above, communication from some process p to p' was encoded as an atomic proposition a such that $a \in \mathcal{O}(p)$ and $a \in \mathcal{I}(p')$. In the fault-tolerance encoding, we split this connection into a sending part $a_s \in \mathcal{O}(p)$ and a receiving part $a_r \in \mathcal{I}(p')$ where $a_r \in \mathcal{O}(p_{env})$ is a proposition controlled by the environment. To relate a_s and a_r, we add the assumption $\square(a_s \leftrightarrow a_r)$ to the LTL specification. This encoding uses more atomic propositions and additional LTL constraints but is otherwise equivalent to the one presented before.

This increased flexibility, that is, being able to specify communication using temporal logic, allows us to express unreliable communication. For example, using the assumption $\square(a_s \leftrightarrow \bigcirc a_r)$ specifies a delay of one time step on the receiver, $\square a_r$ specifies a stuck-at-one fault, and $\top$ specifies a Byzantine fault where the environment takes over the communication. This alone is not enough though: if a process gets such a specification it knows which receiving propositions present actual values and which one is subject to a fault. Thus, the processes are challenged in *multiple architectures*, where each architecture may have a different set of communication faults as well as specifications: depending on the type of failure, the overall system may only be expected to satisfy a weaker property then the original, non-faulty one.

Formally, the fault-tolerant realizability problem is a tuple $\langle \mathcal{A}, \varphi_1, \ldots, \varphi_n \rangle$, where $\mathcal{A}$ is a distributed architecture with the property that every process receives only environment inputs, i.e., $\mathcal{I}(p) \subseteq \mathcal{O}(p_{env})$ for all $p \in P^-$, and $\varphi_1, \ldots, \varphi_n$ are LTL formulas. For Byzantine fault-tolerance, $\varphi_i = \bigwedge_{(s,r)\in R_i} \square(s \leftrightarrow r) \rightarrow \psi_i$ where $R_i \subseteq O \times I$ are the non-faulty communication of architecture i and ψ_i is the LTL specification that should be ensured.

As an example, consider the architecture

$$\begin{aligned}\langle \{p_{env}, p_1, p_2, p_3\}, p_{env}, \{p_1 \mapsto \{a\}, p_2 \mapsto \{a\}, p_3 \mapsto \{b, c\}\},\\ \{p_{env} \mapsto \{a, b, c\}, p_1 \mapsto \{x\}, p_2 \mapsto \{y\}, p_3 \mapsto \{z\}\}\rangle \qquad (2)\end{aligned}$$

with specifications $\varphi_1 = \square((x \leftrightarrow b) \wedge (y \leftrightarrow c)) \rightarrow \psi$, $\varphi_2 = \square(y \leftrightarrow c) \rightarrow \psi$, and $\varphi_3 = \square(x \leftrightarrow b) \rightarrow \psi$. This example specification asserts that ψ holds in all three architectures depicted in Fig. 2, i.e., if either $p_1 \xrightarrow{x} p_3$ or $p_2 \xrightarrow{y} p_3$ fails, but not both of them. Hence, process p_3 cannot know whether the information given via propositions b or c is correct.

Theorem 5 *The HyperLTL realizability problem subsumes the fault-tolerant LTL realizability problem.*

Proof Given $\langle \mathcal{A}, \varphi_1, \ldots, \varphi_n \rangle$, the following HyperLTL formula over inputs $\mathcal{O}(p_{env})$ and outputs $\bigcup_{p\in P^-} \mathcal{O}_p$ is equirealizable:

$$\forall \pi \forall \pi'. \bigwedge_{1\le i\le n} \varphi_i[\pi] \wedge \bigwedge_{p\in P^-} D^{\pi,\pi'}_{\mathcal{I}(p)\mapsto\mathcal{O}(p)}$$

□

4 Deciding HyperLTL synthesis

In this section, we identify fragments of HyperLTL for which the realizability problem is decidable. Our findings are summarized in Table 1.

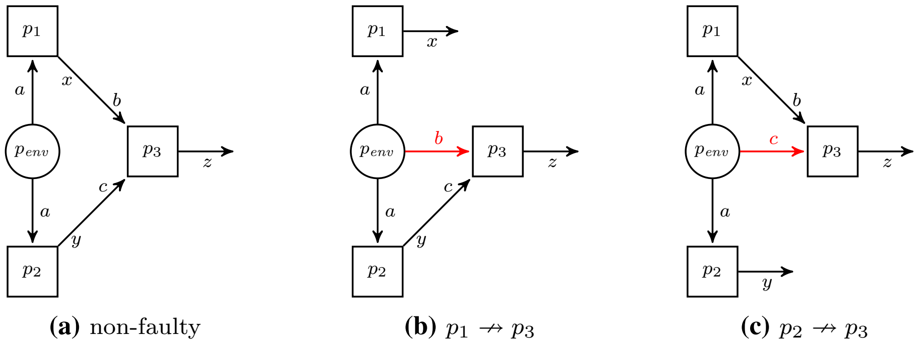


Fig. 2 Visual interpretation of a fault-tolerance specification: on the left is the original (non-faulty) architecture where the communication between $p_1 \to p_3$ and $p_2 \to p_3$ is intact. The two architectures on the right represent the case where either $p_1 \nrightarrow p_3$ or $p_2 \nrightarrow p_3$. In this case, the receiving propositions b and c, respectively, are controlled by the environment. In fault-tolerant synthesis, we search for strategies for processes p_1, p_2, and p_3 such that the specification is satisfied in all architectures

Table 1 Summary of decidability results

$\exists^*$	$\forall^*$	*linear* $\forall^*$	$\exists^*\forall^1$	$\exists^*\forall^{>1}$	$\forall^*\exists^*$
PSPACE-complete	Undecidable	Non-elem.	3EXPTIME	Undecidable	Undecidable

We base our investigation on the structure of the quantifier prefix of the HyperLTL formulas. We call a HyperLTL formula φ (quantifier) *alternation-free* if the quantifier prefix consists solely of either universal or existential quantifiers. We denote the corresponding fragments as the (universal) $\forall^*$ and the (existential) $\exists^*$ fragment, respectively. A HyperLTL formula is in the $\exists^*\forall^*$ fragment, if it starts with arbitrarily many existential quantifiers, followed by arbitrarily many universal quantifiers. Analogously for the $\forall^*\exists^*$ fragment. For a given natural number n, we refer to a bounded number of quantifiers with $\forall^n$, respectively $\exists^n$. The $\forall^1$ realizability problem is equivalent to the LTL realizability problem.

4.1 $\exists^*$ Fragment

We show that the realizability problem for existential HyperLTL is PSPACE-complete. We reduce the realizability problem to the satisfiability problem for bounded one-alternating $\exists^*\forall^2$HyperLTL [18], i.e., finding a trace set T such that $T \vDash \varphi$.

Lemma 1 *An existential HyperLTL formula $\varphi = \exists\pi_1 \cdots \exists\pi_n.\, \psi$ is realizable if, and only if, $\varphi_{sat} := \exists\pi_1 \cdots \exists\pi_n.\, \forall\pi\forall\pi'.\, \psi \wedge D^{\pi,\pi'}_{I\mapsto O}$ is satisfiable.*

Proof Assume $f\colon (2^I)^* \to 2^O$ realizes φ, that is $f \vDash \varphi$. Let $T = traces(f)$ be the set of traces generated by f. It holds that $T \vDash \varphi$ and $T \vDash \forall\pi, \pi'.\, D^{\pi,\pi'}_{I\mapsto O}$. Therefore, T witnesses the satisfiability of φ_{sat}. For the reverse direction, assume that φ_{sat} is satisfiable. Let S be a set of traces that satisfies φ_{sat}. We construct a strategy $f\colon (2^I)^* \to 2^O$ as

$$f(\sigma) = \begin{cases} w_{|\sigma|} \cap O & \text{if } \sigma \text{ is a prefix of some } w|_I \text{ with } w \in S, \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$$

where $w|_I$ denotes the trace restricted to I, meaning that $w_i \cap I$ for all $i \geq 0$. Note that if there are multiple candidates $w \in S$, then $w_{|\sigma|} \cap O$ is equal across all of them due to the required determinism $\forall\pi\forall\pi'.\, D^{\pi,\pi'}_{I\mapsto O}$. By construction, all traces in S are contained in f, and together with $S \vDash \varphi$, it holds that $f \vDash \varphi$ as the sets of sets of traces satisfying the existential formula φ are upward closed. □

Theorem 6 *Realizability of existential HyperLTL specifications is decidable.*

Proof The formula φ_{sat} from Lemma 1 is in the $\exists^*\forall^2$ fragment, for which satisfiability is decidable [18]. □

Corollary 1 *Realizability of existential HyperLTL specifications is* PSPACE*-complete.*

Proof Given an existential HyperLTL formula, we gave a linear reduction to the satisfiability of the $\exists^*\forall^2$ fragment in Lemma 1. The satisfiability problem for a bounded number of universal quantifiers is in PSPACE [18]. Hardness follows from LTL satisfiability [43], which is equivalent to the $\exists^1$ fragment. □

4.2 $\forall^*$ Fragment

In the following, we will use the *distributed synthesis* problem defined before, i.e., the problem whether there is an implementation of processes in a distributed architecture (cf. Fig. 1) that satisfies an LTL formula.

Theorem 7 *The synthesis problem for universal HyperLTL becomes undecidable as soon as we have more than one universal quantifier.*

Proof In the $\forall^*$ fragment (and thus in the $\exists^*\forall^*$ fragment), we can encode a distributed architecture [40], for which LTL synthesis is undecidable. In particular, we can encode the architecture shown in Fig. 1a. This architecture basically specifies c to depend only on a and analogously d on b. That can be encoded by $D^{\pi,\pi'}_{\{a\}\mapsto\{c\}}$ and $D^{\pi,\pi'}_{\{b\}\mapsto\{d\}}$. The LTL synthesis problem for this architecture is undecidable [40], i.e., given an LTL formula over $I = \{a, b\}$ and $O = \{c, d\}$, we cannot automatically construct processes p_1 and p_2 that realize the formula. □

4.3 Linear $\forall^*$ fragment

As we have already seen in Theorem 2, we can represent distributed architectures in HyperLTL by using the dependency formulas $D^{\pi,\pi'}_{I\mapsto O}$. In this section, we do the reverse direction. Given a HyperLTL formula φ, we try to find an equivalent formula which has the distributed form $\forall\pi.\,\forall\pi'.\,\varphi' \wedge (dep)$ where φ' only uses the path variable π and where (dep) is a conjunction of dependency formulas. These formulas can then be translated to distributed architectures on which we can use the already existent techniques to deal with the synthesis problem. In particular, we can check whether the resulting distributed synthesis problem has an information fork. If this is not the case, the problem is decidable [27].

For finding the distributed form, we need to find the "LTL-part" φ' of the HyperLTL property φ. We call this *collapsing* φ to φ'. This transformation collapses the universal quantifier into a single one and renames the path variables accordingly. For example, $\forall\pi_1\forall\pi_2.\,\Box a_{\pi_1} \vee \Box a_{\pi_2}$ is transformed into an equivalent $\forall^1$ formula $\forall\pi.\,\Box a_\pi \vee \Box a_\pi$. However, this transformation does not always produce equivalent formulas as $\forall\pi_1\forall\pi_2.\,\Box(a_{\pi_1} \leftrightarrow$

$a_{\pi_2})$ is not equivalent to its collapsed form $\forall \pi. \square(a_\pi \leftrightarrow a_\pi)$. Let φ be $\forall \pi_1 \cdots \forall \pi_n. \psi$. We define the collapsed formula of φ as $collapse(\varphi) := \forall \pi. \psi[\pi_1 \mapsto \pi][\pi_2 \mapsto \pi] \ldots [\pi_n \mapsto \pi]$ where $\psi[\pi_i \mapsto \pi]$ replaces all occurrences of π_i in ψ with π. Although the collapsed term is not always equivalent to the original formula, we can use it as an indicator whether it is possible at all to express a universal formula with only one quantifier as stated in the following lemma.

Lemma 2 *Either $\varphi \equiv collapse(\varphi)$ or φ has no equivalent $\forall^1$ formula.*

Proof Suppose there is some $\psi \in \forall^1$ with $\psi \equiv \varphi$. We show that $\psi \equiv collapse(\varphi)$. Let T be an arbitrary set of traces. Let $\mathcal{T} = \{\{w\} \mid w \in T\}$. Because $\psi \in \forall^1$, $T \vDash \psi$ is equivalent to $\forall T' \in \mathcal{T}. T' \vDash \psi$, which is by assumption equivalent to $\forall T' \in \mathcal{T}. T' \vDash \varphi$. Now, φ operates on singleton trace sets only. This means that all quantified paths have to be the same, which yields that we can use the same path variable for all of them. So $\forall T' \in \mathcal{T}. T' \vDash \varphi \leftrightarrow T' \vDash collapse(\varphi)$ that is again equivalent to $T \vDash collapse(\varphi)$. Because $\psi \equiv collapse(\varphi)$ and $\psi \equiv \varphi$ it holds that $\varphi \equiv collapse(\varphi)$. □

Now that we have the φ'-part of the distributive form, we need to find the variable dependencies. More precisely, given a formula φ, we seek for variable dependencies of the form $D^{\pi,\pi'}_{J \mapsto \{o\}}$ with $J \subseteq I$ for every output $o \in O$. These $J's$ can be brute-forced. For doing so, we just check for each case the equivalence between φ and $\forall \pi. \forall \pi'. collapse(\varphi) \wedge (dep)$ where (dep) is the conjunction of all $D^{\pi,\pi'}_{J \mapsto \{o\}}$. If this is the case and furthermore, the $D^{\pi,\pi'}_{J_i \mapsto \{o_i\}}$ constraints can be ordered such that $J_i \subseteq J_{i+1}$ for all i, we have a linear architecture. Linear architectures are architectures without information fork. Thus, they are decidable. We define the linear fragment to encompass exactly these linear architectures. All in all, there are three steps to check whether φ is in the linear fragment:

1. First, we have to add input-determinism to the formula $\varphi_{det} := \forall \pi. \forall \pi'. \varphi \wedge D^{\pi,\pi'}_{I \mapsto O}$. This preserves realizability as strategies are input-deterministic.
2. Find for each output variable $o_i \in O$ possible sets of variables J_i on which the o_i depend, such that $J_i \subseteq J_{i+1}$. To check whether the choice of J's is correct, we test if $\forall \pi. \forall \pi'. collapse(\varphi) \wedge \bigwedge_{o_i \in O} D^{\pi,\pi'}_{J_i \mapsto \{o_i\}}$ is equivalent to φ_{det}. This equivalence check is decidable as both formulas are in the universal fragment [18].
3. Finally, we construct the corresponding distributed realizability problem with linear architecture. We define the distributed architecture $\mathcal{A} = \langle P, p_{env}, \mathcal{I}, \mathcal{O} \rangle$ with $P = \{p_i \mid o_i \in O\} \cup \{p_{env}\}$, $\mathcal{I}(p_i) = J_i$, $\mathcal{O}(p_i) = \{o_i\}$, and $\mathcal{O}(p_{env}) = I$. The LTL specification for the distributed synthesis problem is $collapse(\varphi)$

Definition 2 *(linear fragment of $\forall^*$)* In the context of a synthesis problem with inputs I and outputs O, a formula φ is in the linear fragment of $\forall^*$ iff for all $o_i \in O$ there is a $J_i \subseteq I$ such that $\forall \pi. \forall \pi'. \varphi \wedge D^{\pi,\pi'}_{I \mapsto O} \equiv \forall \pi. \forall \pi'. collapse(\varphi) \wedge \bigwedge_{o_i \in O} D^{\pi,\pi'}_{J_i \mapsto \{o_i\}}$ and $J_i \subseteq J_{i+1}$ for all i.

Note that each $\forall^1$ formula φ (or φ is collapsible to a $\forall^1$ formula) is in the linear fragment because we can set all $J_i = I$ and additionally $collapse(\varphi) = \varphi$ holds.

An example of a formula in the linear fragment of $\forall^*$ is $\varphi = \forall \pi, \pi'. D^{\pi,\pi'}_{\{a\} \mapsto \{c\}} \wedge \square(c_\pi \leftrightarrow d_\pi) \wedge \square(b_\pi \leftrightarrow \bigcirc e_\pi)$ with $I = \{a, b\}$ and $O = \{c, d, e\}$. The corresponding formula asserting input-deterministism is $\forall \pi. \forall \pi'. \varphi_{det} = \varphi \wedge D^{\pi,\pi'}_{I \mapsto O}$. One possible choice of J's is $\{a, b\}$ for c, $\{a\}$ for d and $\{a, b\}$ for e. Note, that one can use either $\{a, b\}$ or $\{a\}$ for c as $\forall \pi. \forall \pi'. D^{\pi,\pi'}_{\{a\} \mapsto \{d\}} \wedge (c_\pi \leftrightarrow d_\pi)$ implies $\forall \pi. \forall \pi'. D^{\pi,\pi'}_{\{a\} \mapsto \{c\}}$. However, the apparent alternative

$\{b\}$ for e would yield an undecidable architecture with information fork. It holds that φ_{det} and $\forall\pi.\forall\pi'.\ collapse(\varphi) \wedge D^{\pi,\pi'}_{\{a,b\}\mapsto\{c\}} \wedge D^{\pi,\pi'}_{\{a\}\mapsto\{d\}} \wedge D^{\pi,\pi'}_{\{a,b\}\mapsto\{e\}}$ are equivalent and, thus, that φ is in the linear fragment.

Theorem 8 *The realizability of the linear fragment of HyperLTL is decidable.*

Proof It holds that $\forall\pi.\forall\pi'.\ \varphi \wedge D^{\pi,\pi'}_{I\mapsto O} \equiv \forall\pi.\forall\pi'.\ collapse(\varphi) \wedge \bigwedge_{o_i\in O} D^{\pi,\pi'}_{J_i\mapsto\{o_i\}}$ for some J_i's. The LTL distributed realizability problem for $collapse(\varphi)$ in the constructed architecture A is equivalent to the HyperLTL realizability of φ as the architecture $\mathcal{A}$ represents exactly the input-determinism represented by formula $\bigwedge_{o_i\in O} D^{\pi,\pi'}_{J_i\mapsto\{o_i\}}$. The architecture is linear and, thus, the realizability problem is decidable. □

Corollary 2 *The realizability problem of the linear fragment of HyperLTL can be checked in non-elementary time.*

The reason for this is that solving the distributed synthesis problem takes non-elementary time in the amount of variables. Surprisingly, this runtime is not (more than linearly) dependent on the amount of quantifiers as they are directly reduced to two.

4.4 $\exists^*\forall^1$ Fragment

In this fragment, we consider arbitrary many existential trace quantifiers followed by a single universal trace quantifier. This fragment turns out to be still decidable. We solve the realizability problem for this fragment by reducing it to a decidable fragment of the distributed realizability problem for LTL.

Theorem 9 *Realizability of* $\exists^*\forall^1$*HyperLTL specifications is decidable.*

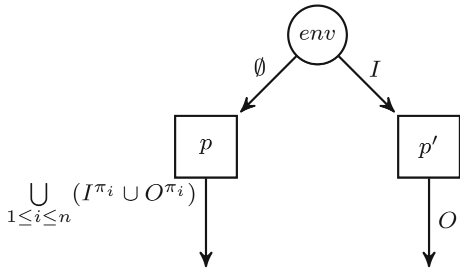
Proof Let φ be $\exists\pi_1\cdots\exists\pi_n.\forall\pi'.\ \psi$. We reduce the realizability problem of φ to the distributed realizability problem for LTL. Intuitively, we use a two-process distributed architecture where the first process p is supposed to produce the traces for the leading existential quantification and the second process p' represents the realizing strategy. The architecture is depicted in Fig. 3.

For every existential trace quantifier π, we introduce a copy of every atomic proposition for the distributed realizability problem, written a^π for $a \in \mathrm{AP}$. We use the same notation for sets of atomic propositions, e.g., $I^\pi = \{i^\pi \mid i \in I\}$. Process p has no inputs, thus, produces only a single trace, and it controls the outputs $\bigcup_{1\le i\le n} (I^{\pi_i} \cup O^{\pi_i})$. Using an appropriate valuation of its outputs, process p selects the paths in the strategy tree corresponding to the existential trace quantifiers $\exists\pi_1\cdots\exists\pi_n$. Thus, those output propositions of process p have to encode an actual path in the strategy tree produced by p'. To ensure this, we add the LTL constraint $\square(I^{\pi_i} = I) \rightarrow \square(O^{\pi_i} = O)$ that asserts that if the inputs correspond to some path in the strategy tree, the outputs on those paths have to be the same. The resulting architecture $\mathcal{A}_\varphi$ is

$$\langle\{p_{env}, p, p'\}, p_{env}, \{p \mapsto \emptyset, p' \mapsto I\},$$
$$\{p_{env} \mapsto I, p \mapsto \bigcup_{1\le i\le n} (I^{\pi_i} \cup O^{\pi_i}), p' \mapsto O\}\rangle.$$

It is easy to verify that $\mathcal{A}_\varphi$ does not contain an information fork, thus the realizability problem is decidable [27]. The LTL specification θ is $\psi \wedge \bigwedge_{1\le i\le n} \square(I^{\pi_i} = I) \rightarrow \square(O^{\pi_i} = O)$ where we replace every a_π by a^π for existential traces and $a_{\pi'}$ to a in ψ. The implementation of

Fig. 3 Visualization of the architecture used in the $\exists^*\forall^1$ reduction in the proof of Theorem 9



process p' (if it exists) is a realizing strategy for the HyperLTL formula (process p producing witnesses for the $\exists$ quantifiers): Assume that there are realizing strategies for $\langle \mathcal{A}_\varphi, \theta\rangle$, i.e., $f_p\colon (2^\emptyset)^* \to 2^{\bigcup_{1\le i\le n}(I^{\pi_i}\cup O^{\pi_i})}$ and $f_{p'}\colon (2^I)^* \to 2^O$. $f_{p'}$ is a realizing strategy for φ as well: By the HyperLTL semantics, we have to show that there is a trace assignment $\Pi\colon \mathcal{V}_\exists \to$ *traces*$(f_{p'})$ such that for all $t \in$ *traces*$(f_{p'})$ it holds that $\Pi[\pi' \to t] \vDash_{traces(f_{p'})} \psi$. We define Π in the following. Note that *traces*(f_p) is a singleton set and let $t_p \in \left(2^{\bigcup_{1\le i\le n}(I^{\pi_i}\cup O^{\pi_i})}\right)^\omega$ be the corresponding trace. For every $\pi_i \in \{\pi_1,\ldots,\pi_n\}$, we define $\Pi(\pi_i) = t_p|_{I^{\pi_i}}$ where we replace a^{π_i} by a for every $a \in$ AP. This, together with θ shows that ψ holds for every chosen path $t \in$ *traces*$(f_{p'})$ for π'.

Conversely, a model for φ can be used as an implementation of p and p': Let $f\colon (2^I)^* \to 2^O$ be a realizing strategy of φ. We use f as a strategy for p'. We construct the single trace produced by p using the existential trace assignment $\Pi\colon \mathcal{V}_\exists \to$ *traces*(f). Let $t_1,\ldots,t_n \in$ *traces*(f) be the corresponding traces. We construct a single trace t_p by replacing propositions $a \in$ AP by a^{π_i} for every t_i and the subsequent union of the resulting traces (which now have pairwise disjoint propositions). Due to the construction, f_p satisfies $\bigwedge_{1\le i\le n}\Box(I^{\pi_i} = I) \to \Box(O^{\pi_i} = O)$ and thus, the distributed architecture satisfies θ. Hence, the distributed synthesis problem $\langle \mathcal{A}_\varphi, \theta\rangle$ has a solution if, and only if, φ is realizable. $\square$

4.5 ∀*∃* Fragment

The last fragment to consider are formulas in the $\forall^*\exists^*$ fragment. Whereas the $\exists^*\forall^1$ fragment remains decidable, the realizability problem of $\forall^*\exists^*$ turns out to be undecidable even when restricted to only one quantifier of both sorts ($\forall^1\exists^1$).

Theorem 10 *Realizability of* $\forall^*\exists^*$*HyperLTL is undecidable.*

Proof We will prove this theorem by reducing Post's Correspondence Problem (PCP) [41] to synthesizing an $\forall^1\exists^1$ formula. This proof follows the one from [18]. In PCP, we are given two lists α and β of same length which consist of finite words from some alphabet Σ. For example, α, with $\alpha_1 = a$, $\alpha_2 = ab$ and $\alpha_3 = bba$ and β, with $\beta_1 = baa$, $\beta_2 = aa$ and $\beta_3 = bb$. Here α_i denotes the ith element of the list, and $\alpha_{i,j}$ denotes the jth symbol of the ith element. In this example, $\alpha_{3,1}$ corresponds to b. PCP is the problem to find an index sequence $(i_k)_{1\le k\le K}$ with $K \ge 1$ and $1 \le i_k \le n$ for all k, such that $\alpha_{i_1}\ldots\alpha_{i_K} = \beta_{i_1}\ldots\beta_{i_K}$. We denote the finite words of a PCP solution with i_α and i_β, respectively. It is a useful intuition to think of the PCP instance as a set of n domino stones. A possible solution for this PCP instance would be (3, 2, 3, 1), since this stone sequence produces the same word, i.e., $bbaabbbaa = i_\alpha = i_\beta$.

Let a PCP instance with $\Sigma = \{a_1, a_2, \ldots, a_n\}$ and two lists α and β be given. We choose our set of atomic propositions as follows: $AP := I \dot{\cup} O$ with $I := \{i\}$ and $O := (\Sigma \cup \{\dot{a}_1, \dot{a}_2, \ldots, \dot{a}_n\} \cup \#)^2$, where we use the dot symbol to encode that a stone starts at this position of the trace. We write $\tilde{a}$ if we do not care if this symbol is an a or $\dot{a}$ and use $*$ as syntactic sugar for an arbitrary symbol of the alphabet.

We encode the PCP instance into a HyperLTL formula that is realizable if and only if the PCP instance has a solution as follows:

$$\varphi_{reduc} := \varphi_{sol}(\pi) \wedge \forall \pi \exists \pi'.\ \varphi_{rel}(\pi) \rightarrow \varphi_{is++}(\pi, \pi') \wedge \varphi_{start}(\varphi_{stone\&shift}(\pi, \pi'), \pi)$$

- $\varphi_{\mathrm{sol}}(\pi) := \Box i_\pi \rightarrow (\bigvee_{i=1}^{n} (\dot{a}_i, \dot{a}_i)_\pi) \wedge (\bigvee_{i=1}^{n} (\tilde{a}_i, \tilde{a}_i)_\pi)\, \mathcal{U} \Box (\#, \#)_\pi$. Here, we associate the left part of the pairs with the upper (α) part of the domino stones and the right part with the lower (β) part of the domino stones. And only taking the left side of the pairs along the path π we should get i_α, respectively for i_β. Therefore, this formula essentially states that the path which has $\Box i$ encodes our *solution* and has $i_\alpha = i_\beta$. The # symbols at the end are placeholders for ensuring that the sequence has a finite size.
- $\varphi_{rel}(\pi) := \neg i_\pi\, \mathcal{U} \Box i_\pi$. This defines the set of *relevant* traces trough our strategy tree. They can be imagined being *parallel* to each other or thought of as the solution trace $\Box i$ but with some additional $\neg i$ in front.
- $\varphi_{is++}(\pi, \pi') := (\neg i_\pi \wedge \neg i_{\pi'})\, \mathcal{U} (\Box i_\pi \wedge \neg i_{\pi'} \wedge \bigcirc \Box i_{\pi'})$. This defines that a trace π' is a successor of π. It essentially states that π' has exactly one more $\neg i$ at the beginning than π, i.e., it is the *next* parallel trace.
- Essentially, we now want π' to be exactly π but with the first stone removed and the rest shifted to the front. This can be best illustrated with the example from above. The full sequence at the trace $\Box i$ represents the solution with the outputs

  $$(\dot{b}, \dot{b})(b, b)(a, \dot{a})(\dot{a}, a)(b, \dot{b})(\dot{b}, b)(b, \dot{b})(a, a)(\dot{a}, a)(\#, \#)(\#, \#)\ldots$$

  The next trace, which is the one with $\neg i$ in front and then $\bigcirc \Box i$ is

  $$(\dot{a}, \dot{a})(b, a)(\dot{b}, \dot{b})(b, b)(a, \dot{b})(\dot{a}, a)(\#, a)(\#, \#)(\#, \#)\ldots$$

  Note that now the symbols are not aligned any more because the first stone did not have an equal length of the upper and lower part. We continue this sequence by removing the next stones:

  $$(\dot{b}, \dot{b})(b, b)(a, \dot{b})(\dot{a}, a)(\#, a)(\#, \#)(\#, \#)\ldots$$
  $$(\dot{a}, \dot{b})(\#, a)(\#, a)(\#, \#)(\#, \#)\ldots$$
  $$(\#, \#)(\#, \#)\ldots$$

  Now, the formula $\varphi_{stone\&shift}(\pi, \pi')$ encodes that π' is π with the first stone removed and the rest shifted as illustrated in the example. This can be done with a disjunction over all possible stones. See [18] for more details.
- $\varphi_{start}(\varphi, \pi) := \neg i_\beta\, \mathcal{U} \varphi \wedge \Box i_\pi$ cuts off the irrelevant prefix until φ starts. This irrelevant prefix is exactly the part where the $\neg i$ appear. We are only interested in looking at the $\Box i$ part of the traces because they are not shared by the different *relevant* traces.
- We furthermore assume that only singletons are allowed what can be achieved by a disjunction for each pair of atomic propositions. See [18] as well.

With this construction, we now force by the synthesis algorithm to yield a list of paths which represent the PCP solution step-wise. To get the list of stones used for this PCP solution, we can just examine the first path (with $\square i$) and look at the outputs. As we know that the dots above the letters indicate a new stone, we can slice the sequence i_α to $\alpha_{i_1} \dots \alpha_{i_K}$ and same for i_β. This shows that we can solve PCP if we can solve the synthesis problem for $\forall^1\exists^1$ formulas. □

5 Bounded realizability

We propose an algorithm to synthesize strategies from specifications given in universal HyperLTL by searching for finite generators of realizing strategies. We encode this search as a satisfiability problem for a decidable constraint system.

5.1 Transition systems

A *transition system* $\mathcal{S}$ is a tuple $\langle S, s_0, \tau, l\rangle$ where S is a finite set of states, $s_0 \in S$ is the designated initial state, $\tau\colon S \times 2^I \rightarrow S$ is the transition function, and $l\colon S \rightarrow 2^O$ is the state-labeling or output function. We generalize the transition function to sequences over 2^I by defining $\tau^*\colon (2^I)^* \rightarrow S$ recursively as $\tau^*(\epsilon) = s_0$ and $\tau^*(w_0 \cdots w_{n-1} w_n) = \tau(\tau^*(w_0 \cdots w_{n-1}), w_n)$ for $w_0 \cdots w_{n-1} w_n \in (2^I)^+$. A transition system $\mathcal{S}$ *generates* the strategy f if $f(w) = l(\tau^*(w))$ for every $w \in (2^I)^*$. A strategy f is called *finite-state* if there exists a transition system that generates f.

5.2 Overview

We first sketch the synthesis procedure and then proceed with a description of the intermediate steps. Let φ be a universal HyperLTL formula $\forall\pi_1 \cdots \forall\pi_n.\, \psi$. We build the automaton $\mathcal{A}_\psi$ whose language is the set of tuples of traces that satisfy ψ. We then define the acceptance of a transition system $\mathcal{S}$ on $\mathcal{A}_\psi$ by means of the self-composition of $\mathcal{S}$. Lastly, we encode the existence of a transition system accepted by $\mathcal{A}_\psi$ as an SMT constraint system.

Example 1 Throughout this section, we will use the following (simplified) running example. Assume we want to synthesize a system that keeps decisions secret until it is allowed to publish. Thus, our system has three input signals *decision*, indicating whether a decision was made, the secret *value*, and a signal to *publish* results. Furthermore, our system has two outputs, an undisclosed output *internal* that stores the value of the last decision, and a public output *result* that indicates the result. No information about decisions should be inferred until publication. To specify the functionality, we propose the LTL specification

$$\begin{aligned} &\square(\mathit{decision} \rightarrow (\mathit{value} \leftrightarrow \bigcirc\, \mathit{internal})) \\ \wedge\ &\square(\neg \mathit{decision} \rightarrow (\mathit{internal} \leftrightarrow \bigcirc\, \mathit{internal})) \\ \wedge\ &\square(\mathit{publish} \rightarrow \bigcirc(\mathit{internal} \leftrightarrow \mathit{result})). \end{aligned} \tag{3}$$

The solution produced by the LTL synthesis tool BoSy [17], shown in Fig. 4a, clearly violates our intention that results should be secret until publish: Whenever a decision is made, the output *result* changes as well.

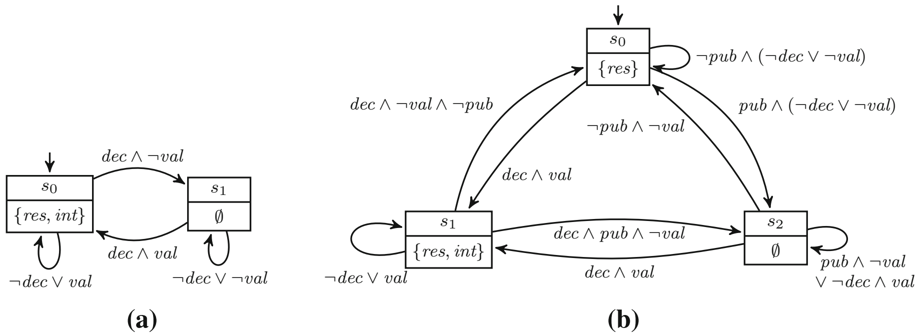


Fig. 4 Synthesized Moore transition systems based on the specification given in Example 1

We formalize the property that no information about the decision can be inferred from *result* until publication as the HyperLTL formula

$$\forall \pi \forall \pi'. (publish_{\pi} \vee publish_{\pi'}) \mathcal{R} (result_{\pi} \leftrightarrow result_{\pi'}). \tag{4}$$

It asserts that for every pair of traces, the *result* signals have to be the same until (if ever) there is a *publish* signal on either trace. A solution satisfying both, the functional specification and the hyperproperty, is shown in Fig. 4b. The system switches states whenever there is a decision with a different value than before and only exposes the decision in case there is a prior publish command.

We proceed with introducing the necessary preliminaries for our algorithm.

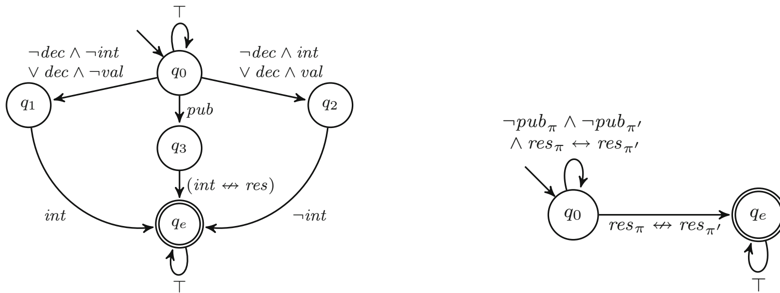
5.3 Automata

A universal co-Büchi automaton $\mathcal{A}$ over finite alphabet Σ is a tuple $\langle Q, q_0, \delta, F\rangle$, where Q is a finite set of states, $q_0 \in Q$ the designated initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ is the set of rejecting states. Given an infinite word $\sigma \in \Sigma^{\omega}$, a run of σ on $\mathcal{A}$ is a finite or infinite path $q_0 q_1 q_2 \cdots \in (Q^* \cup Q^{\omega})$ where for all $i \geq 0$ it holds that $(q_i, \sigma_i, q_{i+1}) \in \delta$. A run is accepting, if it contains only finitely many rejecting states, i.e., either the run is finite or there exists a $i \geq 0$ such that for all $j \geq i$ it holds that $q_j \notin F$. $\mathcal{A}$ accepts a word σ, if *all* runs of σ on $\mathcal{A}$ are accepting. The language of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the set $\{\sigma \in \Sigma^{\omega} \mid \mathcal{A} \text{ accepts } \sigma\}$. We represent automata as directed graphs with vertex set Q and a symbolic representation of the transition relation δ as propositional formulas $\mathbb{B}(\Sigma)$. The rejecting states in F are marked by double lines. The automata for the LTL and HyperLTL specifications from Example 1 are depicted in Fig. 5.

5.4 Run graph

The run graph of a transition system $\mathcal{S} = \langle S, s_0, \tau, l\rangle$ on a universal co-Büchi automaton $\mathcal{A} = \langle Q, q_0, \delta, F\rangle$ is a directed graph $\langle V, E\rangle$ where $V = S \times Q$ is the set of vertices and $E \subseteq V \times V$ is the edge relation with

$$((s, q), (s', q')) \in E \text{ iff}$$
$$\exists i \in 2^I. \exists o \in 2^O. (\tau(s, i) = s') \wedge (l(s) = o) \wedge (q, i \cup o, q') \in \delta.$$



(a) Automaton accepting the language defined by the LTL formula in (3)

(b) Automaton accepting the language defined by the HyperLTL formula in (4)

Fig. 5 Universal co-Büchi automata recognizing the languages from Example 1

A run graph is accepting if every path (starting at the initial vertex (s_0, q_0)) has only finitely many visits of rejecting states. To show acceptance, we annotate every reachable node in the run graph with a natural number m, such that any path, starting in the initial state, contains less than m visits of rejecting states. Such an annotation exists if, and only if, the run graph is accepting [28].

5.5 Self-composition

The model checking of universal HyperLTL formulas [26] is based on self-composition. Let prj_i be the projection to the i-th element of a tuple. Let *zip* denote the usual function that maps an n-tuple of sequences to a single sequence of n-tuples, for example, $zip([1, 2, 3], [4, 5, 6]) = [(1, 4), (2, 5), (3, 6)]$, and let *unzip* denote its inverse. The transition system $\mathcal{S}^n$ is the n-fold self-composition of $\mathcal{S} = \langle S, s_0, \tau, l \rangle$, if $\mathcal{S}^n = \langle S^n, s_0^n, \tau', l^n \rangle$ and for all $s, s' \in S^n$, $\alpha \in (2^I)^n$, and $\beta \in (2^O)^n$ we have that $\tau'(s, \alpha) = s'$ and $l^n(s) = \beta$ iff for all $1 \leq i \leq n$, it holds that $\tau(prj_i(s), prj_i(\alpha)) = prj_i(s')$ and $l(prj_i(s)) = prj_i(\beta)$. If T is the set of traces generated by $\mathcal{S}$, then $\{zip(t_1, \ldots, t_n) \mid t_1, \ldots, t_n \in T\}$ is the set of traces generated by $\mathcal{S}^n$.

We construct the universal co-Büchi automaton $\mathcal{A}_\psi$ such that the language of $\mathcal{A}_\psi$ is the set of words w such that $unzip(w) = \Pi$ and $\Pi \vDash_\emptyset \psi$, i.e., the tuples of traces that satisfy ψ. We get this automaton by dualizing the non-deterministic Büchi automaton for $\neg\psi$ [28], i.e., changing the branching from non-deterministic to universal and the acceptance condition from Büchi to co-Büchi. Hence, $\mathcal{S}$ satisfies a universal HyperLTL formula $\varphi = \forall\pi_1 \cdots \forall\pi_n. \psi$ if the traces generated by self-composition $\mathcal{S}^n$ are a subset of $\mathcal{L}(\mathcal{A}_\psi)$.

Lemma 3 *A transition system $\mathcal{S}$ satisfies the universal HyperLTL formula $\varphi = \forall\pi_1 \cdots \forall\pi_n. \psi$, if, and only if, the run graph of $\mathcal{S}^n$ on $\mathcal{A}_\psi$ is accepting.*

5.6 Synthesis

Let $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ and $\mathcal{A}_\psi = \langle Q, q_0, \delta, F \rangle$. We encode the synthesis problem as an SMT constraint system. Therefore, we use uninterpreted function symbols to encode the transition system and the annotation. For the transition system, those functions are the transition function $\tau : S \times 2^I \rightarrow S$ and the labeling function $l : S \rightarrow 2^O$. The annotation is split into two parts,

a reachability constraint $\lambda^{\mathbb{B}} : S^n \times Q \to \mathbb{B}$ indicating whether a vertex in the run graph is reachable and a counter $\lambda^{\#} : S^n \times Q \to \mathbb{N}$ that maps every reachable vertex to the maximal number of rejecting vertices visited by any path starting in the initial vertex. The resulting constraint asserts that there is a transition system with an accepting run graph.

$$\forall s, s' \in S^n.\, \forall q, q' \in Q.\, \forall i \in (2^I)^n.$$
$$\left(\lambda^{\mathbb{B}}(s, q) \wedge \tau'(s, i) = s' \wedge (q, i \cup l(s), q') \in \delta\right) \to \lambda^{\mathbb{B}}(s', q') \wedge \lambda^{\#}(s', q') \trianglerighteq \lambda^{\#}(s, q)$$

where $\trianglerighteq$ is $>$ if $q' \in F$ and $\geq$ otherwise.

Theorem 11 *The constraint system is satisfiable with bound b if, and only if, there is a transition system $\mathcal{S}$ of size b that realizes the HyperLTL formula.*

We extract a realizing implementation by asking the satisfiability solver to generate a model for the uninterpreted functions that encode the transition system.

6 Bounded unrealizability

So far, we focused on the positive case, providing an algorithm for finding small solutions, if they exist. In this section, we shift to the case of detecting if a universal HyperLTL formula is unrealizable. We adapt the definition of counterexamples to realizability for LTL [29] to HyperLTL in the following. Let φ be a universal HyperLTL formula $\forall \pi_1 \cdots \forall \pi_n.\, \psi$ over inputs I and outputs O, a *counterexample to realizability* is a set of input traces $\mathcal{P} \subseteq (2^I)^\omega$ such that for every strategy $f \colon (2^I)^* \to 2^O$ the labeled traces $\mathcal{P}^f \subseteq (2^{I \cup O})^\omega$ satisfy $\neg\varphi = \exists \pi_1 \cdots \exists \pi_n.\, \neg\psi$.

Proposition 1 *A universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n.\, \psi$ is unrealizable if, and only if, there is a counterexample $\mathcal{P}$ to realizability.*

Proof Let $\mathcal{P}$ be a counterexample to realizability. Assume for contradiction that φ is realizable by a strategy f. By definition of $\mathcal{P}$, we know that $\mathcal{P}^f \vDash \exists \pi_1 \cdots \exists \pi_n.\, \neg\psi$. This means that there exists an assignment $\Pi_{\mathcal{P}} \colon \mathcal{V} \to \mathcal{P}^f$ with $\Pi_{\mathcal{P}} \vDash_{\mathcal{P}^f} \neg\psi$, which is equivalent to $\Pi_{\mathcal{P}} \nvDash_{\mathcal{P}^f} \psi$. Therefore, not all assignments $\Pi : \mathcal{V} \to \mathcal{P}^f$ satisfy $\Pi \vDash_{\mathcal{P}^f} \psi$, which implies that $\mathcal{P}^f \nvDash \forall \pi_1 \cdots \forall \pi_n.\, \psi = \varphi$. Hence, $f \nvDash \varphi$, which concludes the contradiction.

Let φ be unrealizable. We show that the set $\mathcal{P} = (2^I)^\omega$ is a counterexample to realizability. Let $f \colon (2^I)^* \to 2^O$ be an arbitrary strategy, and let $\mathcal{P}^f$ be the corresponding set of labeled traces. From the unrealizability of φ, we now that $f \nvDash \forall \pi_1 \cdots \forall \pi_n.\, \psi$. Thus, there exists a trace assignment $\Pi_{\mathcal{P}} \colon \mathcal{V} \to \mathcal{P}^f$ with $\Pi_{\mathcal{P}} \vDash_{\mathcal{P}^f} \neg\psi$, which is equivalent to $\mathcal{P} \vDash \exists \pi_1 \cdots \exists \pi_n.\, \neg\psi$. □

Despite being independent of strategy trees, there are in many cases finite representations of $\mathcal{P}$. Consider, for example, the unrealizable specification $\varphi_1 = \forall \pi \forall \pi'.\, \Diamond(i_\pi \leftrightarrow i_{\pi'})$, where the set $\mathcal{P}_1 = \{\emptyset^\omega, \{i\}^\omega\}$ is a counterexample to realizability. As a second example, consider $\varphi_2 = \forall \pi \forall \pi'.\, \Box(o_\pi \leftrightarrow o_{\pi'}) \wedge \Box(i_\pi \leftrightarrow \bigcirc o_\pi)$ with conflicting requirements on o. $\mathcal{P}_1$ is a counterexample to realizability for φ_2 as well: By choosing a different valuation of i in the first step of $\mathcal{P}_1$, the system is forced to either react with different valuations of o (violating the first conjunct), or not correctly repeating the initial value of i (violating the second conjunct).

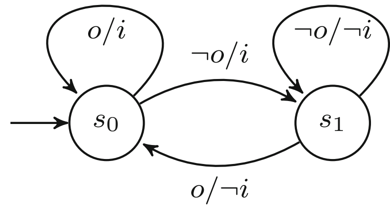


Fig. 6 Counterexample strategy for (5)

There are, however, already linear specifications where the set of counterexample traces is not finite and depends on the strategy tree [30]. For example, the specification

$$\forall \pi. \diamondsuit (o_\pi \leftrightarrow \bigcirc i_\pi) \tag{5}$$

is unrealizable as the system cannot predict future values of the environment. There is no finite set of traces witnessing this: For every finite set of traces, there is a strategy tree such that $\diamondsuit(o_\pi \leftrightarrow \bigcirc i_\pi)$ holds on every such trace. On the other hand, there is a simple *counterexample strategy*, that is a strategy that observes output sequences and produces inputs, depicted in Fig. 6. In this example, the counterexample strategy inverts the outputs given by the system, thus it is guaranteed that $\square(o \not\leftrightarrow \bigcirc i)$ for every system strategy.

We combine those two approaches, selecting counterexample traces and using strategic behavior. A k-counterexample strategy for $\forall^n$HyperLTL observes k output sequences and produces k inputs, where k is a new parameter. We require that k is at least the number of universal quantifiers n. The counterexample strategy is winning if (1) either the traces given by the system player do not correspond to a strategy, or (2) the body of the HyperLTL formula is violated for any n subset of k traces. Regarding property (1), consider the two traces where the system player produces different outputs initially. Clearly, those two traces cannot be generated by any system strategy since the initial state (root labeling) is fixed.

We reduce the search for a k-counterexample strategy to LTL synthesis. For every atomic proposition $a \in \mathrm{AP}$, we produce k copies $a^1, \ldots, a^k$. We use the same notation for sets of atomic propositions, e.g., $I^j = \{i^j \mid i \in I\}$ for $1 \leq j \leq k$. The search for a k-counterexample strategy can be reduced to LTL synthesis using k-tuple input propositions O^k, k-tuple output propositions I^k, and the formula

$$\mathit{strategic}(I^k, O^k) \rightarrow \bigvee_{P \subseteq \{1,\ldots,k\} \text{ with } |P|=n} \neg\psi[P],$$

where $\psi[P]$ denotes the replacement of a_{π_i} by the P_ith position of the combined input/output k-tuple. The formula $\mathit{strategic}(I^k, O^k)$ enforces that the behavior of the system player is strategic and is defined as

$$\bigwedge_{1 \leq j_1 < j_2 \leq k} \left(\bigvee_{i \in I} (i^{j_1} \not\leftrightarrow i^{j_2}) \right) \mathcal{R} \left(\bigwedge_{o \in O} (o^{j_1} \leftrightarrow o^{j_2}) \right).$$

This is an instance of the formula (sym) given in Sect. 3.

Theorem 12 *A universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n. \psi$ is unrealizable if there is a k-counterexample strategy for some $k \geq n$.*

Proof Fix φ and let $f_{cex} : (2^{O^k})^+ \rightarrow 2^{I^k}$ be a k-counterexample strategy. Assume for contradiction that $f : (2^I)^* \rightarrow 2^O$ is a strategy realizing φ. Let $f^k : (2^{I^k})^* \rightarrow 2^{O^k}$ be the

strategy that represents the k-fold self-composition of f (adapting atomic propositions as described earlier). By combining f and f_{cex}, we get an infinite sequence $t \in (2^{I^k \cup O^k})^\omega$: $t_0 = f(\epsilon) \cup f_{cex}(f(\epsilon)), t_1 = f(f_{cex}(f(\epsilon))) \cup f_{cex}(f(f_{cex}(f(\epsilon)))), \ldots$ This sequence represents a k-tuple $cex_k = (I \cup O)^k$. As f^k satisfies $strategic(I^k, O^k)$, there is a n-tuple cex_n build from elements of cex_k such that for the corresponding trace assignment Π it holds that $\Pi \vDash \neg\psi$. This contradicts our assumption that f is a realizing strategy. □

7 Evaluation

We implemented a prototype synthesis tool, called BoSyHyper,[1] for universal HyperLTL based on the bounded synthesis algorithm described in Sect. 5. Furthermore, we implemented the search for counterexamples proposed in Sect. 6. Thus, BoSyHyper is able to characterize realizability and unrealizability of universal HyperLTL formulas.

We base our implementation on the LTL synthesis tool BoSy [17]. For efficiency, we split the specifications into two parts, a part containing the linear (LTL) specification, and a part containing the hyperproperty given as HyperLTL formula. Consequently, we build two constraint systems, one using the standard bounded synthesis approach [28] and one using the approach described in Sect. 5. Before solving, those constraints are combined into a single SMT query. This results in a much more concise constraint system compared to the one where the complete specification is interpreted as a HyperLTL formula. For solving the SMT queries, we use the Z3 solver [12]. We continue by describing the benchmarks used in our experiments.

7.1 Symmetric mutual exclusion

Our first example demonstrates the ability to specify symmetry in HyperLTL for a simple mutual exclusion protocol. Let r_1 and r_2 be input signals representing mutually exclusive *requests* to a critical section and g_1/g_2 the respective grants to enter the section. Every request should be answered eventually $\square(r_i \rightarrow \diamondsuit g_i)$ for $i \in \{1, 2\}$, but not at the same time $\square\neg(g_1 \wedge g_2)$. The minimal LTL solution is depicted in Fig. 7a. It is well known that no mutex protocol can ensure perfect symmetry [37], thus when adding the symmetry constraint specified by the HyperLTL formula $\forall\pi\forall\pi'.\, (r_{1_\pi} \not\leftrightarrow r_{2_{\pi'}})\ \mathcal{R}\ (g_{1_\pi} \leftrightarrow g_{2_{\pi'}})$ the formula becomes unrealizable. Our tool produces the counterexample shown in Fig. 7b. By adding another input signal *tie*, that breaks the symmetry in case of simultaneous requests and modifying the symmetry constraint $\forall\pi\forall\pi'.\, ((r_{1_\pi} \not\leftrightarrow r_{2_{\pi'}}) \vee (tie_\pi \not\leftrightarrow \neg tie_{\pi'}))\mathcal{R}(g_{1_\pi} \leftrightarrow g_{2_{\pi'}})$ we obtain the solution depicted in Fig. 7c. We further evaluated the same properties on a version that forbids spurious grants, which are reported in Table 2 with prefix *full*.

7.2 Distributed and fault-tolerant systems

In Sect. 4 we presented a reduction of arbitrary distributed architectures to HyperLTL. As an example for our evaluation, consider a setting with two processes, one for *encoding* input signals and one for *decoding*. Both processes can be synthesized simultaneously using a single HyperLTL specification. The (linear) correctness condition states that the decoded signal is always equal to the inputs given to the encoder. Furthermore, the encoder and decoder should

[1] BoSyHyper is available at https://www.react.uni-saarland.de/tools/bosy/.

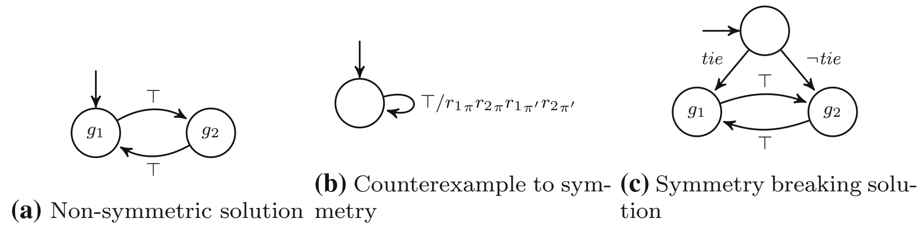


Fig. 7 Synthesized solution of the mutual exclusion protocols

solely depend on the inputs and the encoded signal, respectively. Additionally, we can specify desired properties about the encoding like fault-tolerance [30] or Hamming distance of code words [26]. An example solution for 2 input bits and 3 encoded bits is shown in Fig. 8. For the encoding, we required that for every change in the input, two encoding bits change. The synthesized solution uses a parity bit as the third encoded bit and the encoding and decoding parts are strictly independent. The results are reported in Table 2 where i-j-x means i input bits, j encoded bits, and x represents the property. The property is either tolerance against a single Byzantine signal failure or a guaranteed Hamming distance of code words.

7.3 CAP Theorem

The CAP Theorem due to Brewer [5] states that it is impossible to design a distributed system that provides Consistency, Availability, and Partition tolerance (CAP) simultaneously. This example has been considered before [30] to evaluate a technique that could automatically detect unrealizability. However, when we drop either Consistency, Availability, or Partition tolerance, the corresponding instances (AP, CP, and CA) become realizable, which the previous work was not able to prove. We show that our implementation can show both, unrealizability of CAP and realizability of AP, CP, and CA. In contrast to the previous encoding [30] we are not limited to acyclic architectures.

7.4 Long-term information flow

Previous work on model-checking hyperproperties [26] found that an implementation for the commonly used *I2C* bus protocol could remember input values ad infinitum. For example, it could not be verified that information given to the implementation eventually leaves it, i.e., is forgotten. This is especially unfortunate in high security contexts. We consider a simple bus protocol which is inspired by the widely used *I2C* protocol. Our example protocol has the inputs *send* for initiating a transmission, *in* for the value that should be transferred, and an *ack*nowledgment bit indicating successful transmission. The bus master waits in an *idle* state until a *send* is received. Afterwards, it transmits a header sequence, followed by the value of *in*, waits for an acknowledgement and then indicates *success* or *failure* to the sender before returning to the idle state. We specify the property that the *in*put has no influence on the *data* that is send, which is obviously violated (instance NI1). As a second property, we check that this information leak cannot happen arbitrary long (NI2) for which there is a realizing implementation.

Table 2 Results of BoSyHyper on the benchmarks sets described in Sect. 7

Benchmark	Instance	Result	States		Time [s]	
			Moore	Mealy	Moore	Mealy
Symmetric mutex	Non-sym	Realizable	2	2	1.4	1.3
	Sym	Unrealizable ($k = 2$)	1	1	1.9	2.0
	Tie	Realizable	3	3	1.7	1.6
	Full-non-sym	Realizable	4	4	1.4	1.4
	Full-sym	Unrealizable ($k = 2$)	1	1	4.3	6.2
	Full-tie	Realizable	9	5	1 802.7	5.2
Encoder/decoder	1-2-Hamming-2	Realizable	4	1	1.6	1.3
	1-2-Fault-tolerant	Unrealizable ($k = 2$)	1	–	54.9	–
	1-3-Fault-tolerant	Realizable	4	1	151.7	1.7
	2-2-Hamming-2	Unrealizable ($k = 3$)	–	1	–	10.6
	2-3-Hamming-2	Realizable	16	1	> 1 h	1.5
	2-3-Hamming-3	Unrealizable ($k = 3$)	–	1	–	126.7
CAP Theorem	cap-2-linear	Realizable	8	1	7.0	1.3
	cap-2	Unrealizable ($k = 2$)	1	–	1 823.9	–
	ca-2	Realizable	–	1	–	4.4
	ca-3	Realizable	–	1	–	15.0
	cp-2	Realizable	1	1	1.8	1.6
	cp-3	Realizable	1	1	3.2	10.6
	ap-2	Realizable	–	1	–	2.0
	ap-3	Realizable	–	1	–	43.4
Bus protocol	NI1	Unrealizable ($k = 2$)	1	1	75.2	69.6
	NI2	Realizable	8	8	24.1	33.9
Dining cryptographers	Secrecy	Realizable	–	1	–	82.4

They ran on a machine with a dual-core Core i7, 3.3 GHz, and 16 GB memory

7.5 Dining cryptographers

Recap the dining cryptographers problem introduced earlier. This benchmark is interesting as it contains two types of hyperproperties. First, there is information flow between the three cryptographers, where some secrets (s_{ab}, s_{ac}, s_{bc}) are shared between pairs of cryptographers. In the formalization, we have 4 entities: three processes describing the 3 cryptographers (out_i) and one process computing the result (p_g), i.e., whether the group has paid or not, from out_i. Second, the final result should only disclose whether one of the cryptographers has paid or the NSA. This can be formalized as a indistinguishability property between different executions. For example, when we compare the two traces π and π' where C_a has paid on π and C_b has paid on π', then the outputs of both have to be the same, if their common secret s_{ab} is different on those two traces (while all other secrets s_{ac} and s_{bc} are the same). This ensures that from an outside observer, a flipped output can be either result of a different shared secret or due to the announcement. Lastly, the linear specification asserts that $p_g \leftrightarrow \neg p_{NSA}$.

This question can be encoded as a synthesis problem for HyperLTL. What makes this example particularly interesting is the combination of multiple *information-flow* requirements:

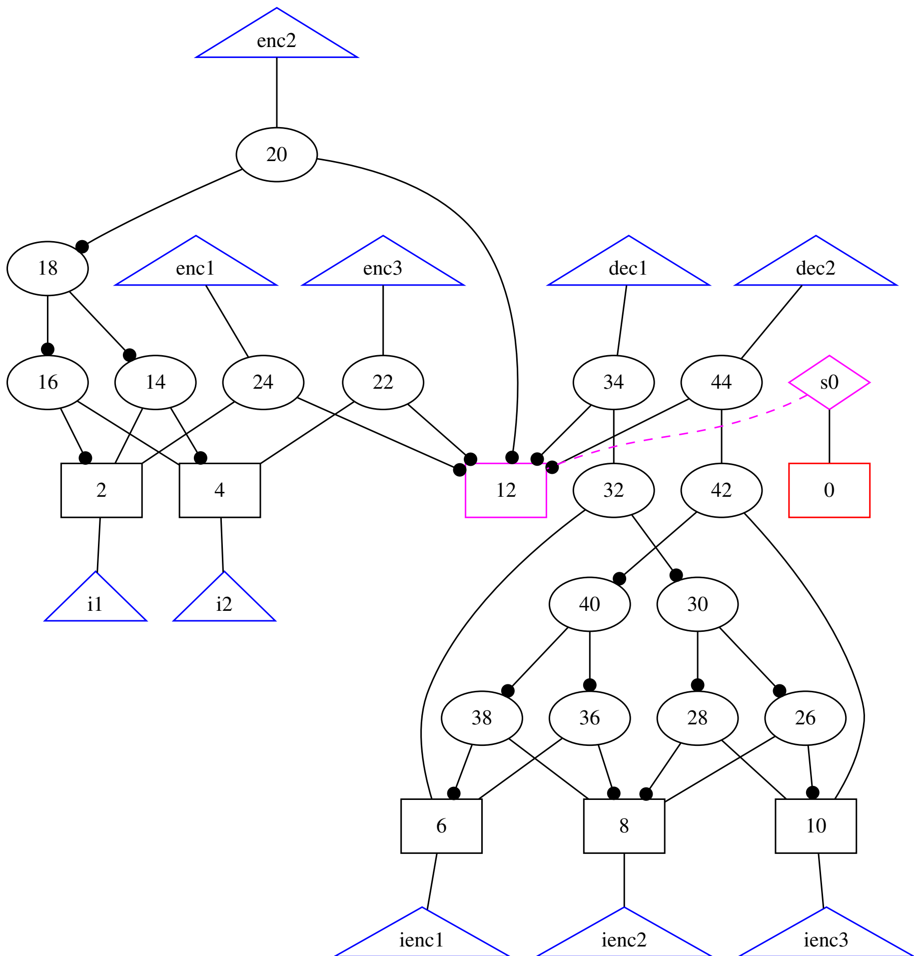


Fig. 8 Representation of the solution for an encoder with 2 input bits and 3 encoded bits as And-Inverter-Graph. The solution is produced by BoSyHyper where the specification is given as a single HyperLTL formula specifying both, the encoder and the decoder, as well as the distributivity constraints. Note that although BoSyHyper produces a global implementation, the implementation is actually distributed as decoder and encoder do not share gates

1. The setting is *distributed*, we have four entities: The three cryptographers (C_a, C_b, and C_c), where each cryptographer shares a secret bit with each other (denoted s_{ab} for the shared secret of C_a and C_b). The fourth entity is the process that receives the output from the cryptographers (out_a, out_b, and out_c) and computes the result whether one of them has paid the bill (output p_g). Figure 9 gives a visual representation of this distributed architecture.
2. The secrecy requirement is formalized as requiring *indistinguishability* between different executions for an outside party (observing out_a, out_b, and out_c). For example, when we compare the two execution traces π and π' where C_a has paid on π and C_b has paid on π'. Then the outputs of both have to be the same, if their common secret s_{ab} is different on those two traces (while all other secrets s_{ac} and s_{bc} are the same). This ensures that

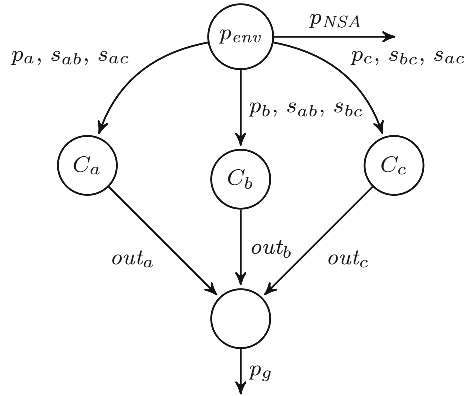


Fig. 9 Architecture for the dining cryptographers

from an outside observer, a flipped output can be either result of a different shared secret or due to the announcement.

We now formalize this as a HyperLTL synthesis problem. The set of atomic propositions is partitioned into environment outputs $I = \{p_{NSA}, p_a, p_b, p_c, s_{ab}, s_{ac}, s_{bc}\}$ and system outputs $O = \{out_a, out_b, out_c, p_g\}$. The functional (LTL) requirements are simple, assuming that exactly one of p_{NSA}, p_a, p_b, and p_c is true, the output of p_g is the negation of p_{NSA}. As an LTL formula

$$\square\Big((p_a \vee p_b \vee p_c \vee p_{NSA}) \wedge mutex(p_a, p_b, p_c, p_{NSA})\Big) \rightarrow \square(p_g \leftrightarrow \neg p_{NSA})$$

The distributed architecture is encoded as a conjunction of HyperLTL formulas ensuring independence of non-observable inputs, i.e.,

$$\begin{aligned}\forall\pi.\,\forall\pi'.D^{\pi,\pi'}_{\{p_a,s_{ab},s_{ac}\}\mapsto\{out_a\}} &\wedge D^{\pi,\pi'}_{\{p_b,s_{ab},s_{bc}\}\mapsto\{out_b\}}\\ \wedge D^{\pi,\pi'}_{\{p_c,s_{ac},s_{bc}\}\mapsto\{out_c\}} &\wedge D^{\pi,\pi'}_{\{out_a,out_b,out_c\}\mapsto\{p_g\}}\end{aligned}$$

Lastly, the indistinguishability (exemplified for C_a and C_b) is formalized as

$$\begin{aligned}\forall\pi\forall\pi'.\square((p^\pi_a \wedge \neg p^{\pi'}_a) \wedge (\neg p^\pi_b \wedge p^{\pi'}_b) \wedge (s^\pi_{ab} \not\leftrightarrow s^{\pi'}_{ab}) \wedge (s^\pi_{bc} \leftrightarrow s^{\pi'}_{bc}) \wedge (s^\pi_{ac} \leftrightarrow s^{\pi'}_{ac})\\ \rightarrow (out^\pi_a \leftrightarrow out^{\pi'}_a) \wedge (out^\pi_b \leftrightarrow out^{\pi'}_b))\end{aligned}$$

for every pair of cryptographers.

Neither LTL synthesis nor its distributed variant can express the combination of those requirements. Our HyperLTL synthesis tool BoSyHyper [20] is able to find a solution to this problem. A closer look in the implementation reveals, that the tool has synthesized the XOR scheme presented in the original solution [6].

7.6 Results

Table 2 reports on the results of the benchmarks. We distinguish between state-labeled (*Moore*) and transition-labeled (*Mealy*) transition systems. Note that the counterexample strategies use the opposite transition system, i.e., a Mealy system strategy corresponds to a state-labeled (Moore) environment strategy. Typically, Mealy strategies are more compact, i.e., need smaller transition systems and this is confirmed by our experiments. BoSyHyper is

able to solve most of the examples, providing realizing implementations or counterexamples. Regrading the unrealizable benchmarks we observe that usually two simultaneously generated paths ($k = 2$) are enough with the exception of the encoder example. Overall the results are encouraging showing that we can solve a variety of instances with non-trivial information flow.

8 Conclusion

In this paper, we have studied the reactive realizability problem for specifications given in the temporal logic HyperLTL. We showed that this problem subsumes various extensions of the LTL realizability problem: synthesis under incomplete information, distributed synthesis, symmetric synthesis, and fault-tolerant synthesis can all be encoded in the synthesis problem of HyperLTL. We gave a complete characterization of the decidable fragments based on the quantifier prefix and, additionally, identified a decidable fragment in the, in general undecidable, universal fragment of HyperLTL. Furthermore, we presented two algorithms to detect realizable and unrealizable HyperLTL specifications, one based on bounding the system implementation and one based on bounding the number of counterexample traces. Our prototype implementation shows that our approach is able to synthesize systems with complex information-flow properties.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, 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 licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence 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. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: Proceedings of CSF, pp. 239–252. IEEE Computer Society (2016). https://doi.org/10.1109/CSF.2016.24
2. Bonakdarpour, B., Finkbeiner, B.: The complexity of monitoring hyperproperties. In: Proceedings of CSF, pp. 162–174. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00019
3. Bonakdarpour, B., Sánchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Proceedings of ISoLA. LNCS, vol. 11245, pp. 8–27. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_2
4. Brett, N., Siddique, U., Bonakdarpour, B.: Rewriting-based runtime verification for alternation-free hyperltl. Proce. TACAS. LNCS **10206**, 77–93 (2017). https://doi.org/10.1007/978-3-662-54580-5_5
5. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of PODC, p. 7. ACM (2000). https://doi.org/10.1145/343477.343502
6. Chaum, D.: Security without identification: transaction systems to make big brother obsolete. Commun. ACM **28**(10), 1030–1044 (1985). https://doi.org/10.1145/4372.4373
7. Clark, D., Hunt, S., Malacaria, P.: Quantified interference for a while language. Electron. Notes Theor. Comput. Sci. **112**, 149–166 (2005). https://doi.org/10.1016/j.entcs.2004.01.018
8. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Proceedings of POST. LNCS, vol. 8414, pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15
9. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393

10. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J.: The hierarchy of hyperlogics. In: Proceedings of LICS, pp. 1–13. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785713
11. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Proceedings of CAV. LNCS, vol. 11561, pp. 121–139. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_7
12. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. Dimitrova, R., Finkbeiner, B.: Synthesis of fault-tolerant distributed systems. In: Proceedings of ATVA. LNCS, vol. 5799, pp. 321–336. Springer (2009). https://doi.org/10.1007/978-3-642-04761-9_24
14. Dimitrova, R., Finkbeiner, B., Kovács, M., Rabe, M.N., Seidl, H.: Model checking information flow in reactive systems. In: Proceedings of VMCAI. LNCS, vol. 7148, pp. 169–185. Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_12
15. Ehlers, R., Finkbeiner, B.: Symmetric synthesis. In: Proceedings of IARCS. LIPIcs, vol. 93, pp. 26:1–26:13. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.FSTTCS.2017.26
16. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of bounded synthesis. Proc. TACAS. LNCS **10205**, 354–370 (2017). https://doi.org/10.1007/978-3-662-54577-5_20
17. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: An experimentation framework for bounded synthesis. In: Proceedings of CAV. LNCS, vol. 10427, pp. 325–332. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_17
18. Finkbeiner, B., Hahn, C.: Deciding hyperproperties. In: Proceedings of CONCUR. LIPIcs, vol. 59, pp. 13:1–13:14. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.13
19. Finkbeiner, B., Hahn, C., Hans, T.: MGHyper: Checking satisfiability of HyperLTL formulas beyond the $\exists^*\forall^*$ fragment. In: Proceedings of ATVA. LNCS, vol. 11138, pp. 521–527. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_31
20. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesizing reactive systems from hyperproperties. In: Proceedings of CAV. LNCS, vol. 10981, pp. 289–306. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_16
21. Finkbeiner, B., Hahn, C., Stenger, M.: Eahyper: Satisfiability, implication, and equivalence checking of hyperproperties. In: Proceedings of CAV. LNCS, vol. 10427, pp. 564–570. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_29
22. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. In: Proceedings of RV. LNCS, vol. 10548, pp. 190–207. Springer (2017). https://doi.org/10.1007/978-3-319-67531-2_12
23. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: A runtime verification tool for temporal hyperproperties. In: Proceedings of TACAS. LNCS, vol. 10806, pp. 194–200. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_11
24. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. Formal Methods Syst. Des. (2019). https://doi.org/10.1007/s10703-019-00334-z
25. Finkbeiner, B., Hahn, C., Torfah, H.: Model checking quantitative hyperproperties. In: Proceedings of CAV. LNCS, vol. 10981, pp. 144–163. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_8
26. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Proceedings of CAV. LNCS, vol. 9206, pp. 30–48. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_3
27. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proceedings of LICS, pp. 321–330. IEEE Computer Society (2005). https://doi.org/10.1109/LICS.2005.53
28. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT **15**(5–6), 519–539 (2013). https://doi.org/10.1007/s10009-012-0228-z
29. Finkbeiner, B., Tentrup, L.: Detecting unrealizable specifications of distributed systems. In: Proceedings of TACAS. LNCS, vol. 8413, pp. 78–92. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_6
30. Finkbeiner, B., Tentrup, L.: Detecting unrealizability of distributed fault-tolerant systems. Log. Methods Comput. Sci. (2015). https://doi.org/10.2168/LMCS-11(3:12)2015
31. Finkbeiner, B., Zimmermann, M.: The first-order logic of hyperproperties. In: Proceedings of STACS. LIPIcs, vol. 66, pp. 30:1–30:14. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.STACS.2017.30
32. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proceedings of S&P, pp. 11–20. IEEE Computer Society (1982). https://doi.org/10.1109/SP.1982.10014
33. Hahn, C.: Algorithms for monitoring hyperproperties. In: Proceedings of RV. LNCS, vol. 11757, pp. 70–90. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_5

34. Hahn, C., Stenger, M., Tentrup, L.: Constraint-based monitoring of hyperproperties. In: Proceedings of TACAS. LNCS, vol. 11428, pp. 115–131. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_7
35. Kupferman, O., Vardi, M.Y.: Synthesis with incomplete information. In: ICTL (1997)
36. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings of FOCS, pp. 531–542. IEEE Computer Society (2005). https://doi.org/10.1109/SFCS.2005.66
37. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems-Safety. Springer, Berlin (1995)
38. McLean, J.: Proving noninterference and functional correctness using traces. J. Comput. Secur. **1**(1), 37–58 (1992). https://doi.org/10.3233/JCS-1992-1103
39. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of POPL, pp. 179–190. ACM Press (1989). https://doi.org/10.1145/75277.75293
40. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proceedings of FOCS. pp. 746–757. IEEE Computer Society (1990). https://doi.org/10.1109/FSCS.1990.89597
41. Post, E.L.: A variant of a recursively unsolvable problem. Bull. Am. Math. Soc. **52**(4), 264–268 (1946)
42. Schewe, S., Finkbeiner, B.: Synthesis of asynchronous systems. In: Proceedings of LOPSTR. LNCS, vol. 4407, pp. 127–142. Springer (2006). https://doi.org/10.1007/978-3-540-71410-1_10
43. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3), 733–749 (1985). https://doi.org/10.1145/3828.3837

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.