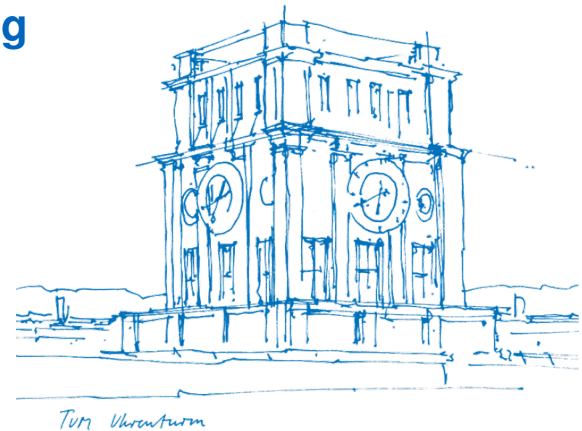
(Hyper)LTL Model Checking

Group phase presentation



Vera Iva, Leonhard Rommel
Dang Khoa Le, Hoang Long Nguyen

Group B
The Hitchhiker's Guide to Hyperproperties
Technical University of Munich

November 13th, 2026

Outline

1. Basic Definitions for Model Checking and Motivation for Hyperproperties
   - System Models, Paths and Traces
   - Noninterference
2. ω-Regular Languages and LTL Model Checking
3. HyperLTL and HyperCTL*
4. Automata-Theoretic Approach for checking HyperCTL*
5. Checking Circuits and Industrial Case Studies
6. Questions?

System Models, Paths and Traces [2]

Kripke Structures

A Kripke structure is a tuple $K = (S, s_0, \delta, AP, L)$:

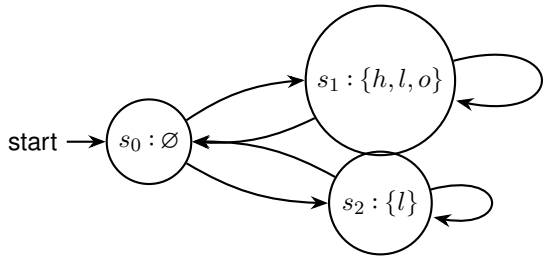
- S: finite set of states, $s_0 \in S$: initial state
- $\delta : S \to 2^S$: transition function
- AP: finite set of atomic propositions
- $L : S \to 2^{AP}$: state labeling function

No Deadlocks / Terminal States: We require $\delta(s) \neq \varnothing$ for all $s \in S$ so every finite execution extends to an infinite path.

System Models, Paths and Traces

Example: System $K_{example}$

Consider a system with 3 atomic propositions $AP = \{h, l, o\}$: **private (high-security) input** h, **public (low-security) input** l and **public output** o.



Here $S = \{s_0, s_1, s_2\}$, $\delta(s_0) = \{s_1, s_2\}$, and $L(s_0) = \varnothing,\ L(s_1) = \{h, l, o\},\ L(s_2) = \{l\}$.

System Models, Paths and Traces

Paths and Traces

Path: Infinite sequence $p = s_0 s_1 s_2 \cdots \in S^\omega$ with $s_{i+1} \in \delta(s_i)$ for all $i \in \mathbb{N}$.

- $Paths(K, s)$: all paths starting in s
- $Paths^*(K, s)$: set of all their suffixes $p[i, \infty] = s_i s_{i+1} \ldots$

Trace: Observable proposition sequence along p:

$$trace(p) = L(s_0)L(s_1)L(s_2) \cdots \in (2^{AP})^\omega$$

System Models, Paths and Traces

Example: Paths and Traces on $K_{example}$

- **Path p (Internal state sequence):** $p = s_0 \to s_1 \to s_1 \to s_1 \ldots$
- **Suffix $p[1, \infty]$ (From step 1 onward):** $p[1, \infty] = s_1 \to s_1 \to s_1 \ldots$
- **Trace $trace(p)$ (Observer's view):** $\varnothing \ \{h, l, o\} \ \{h, l, o\} \ \{h, l, o\} \ldots$

System Models, Paths and Traces

Classical LT Properties

A LT property P over AP is a subset of admissible infinite words: $P \subseteq (2^{AP})^{\omega}$.

- **Satisfaction:** $K \vDash P \iff Traces(K) \subseteq P$ (checked *one trace at a time*).
- **Safety (P_{safe}):** Every bad trace $\sigma \notin P_{\text{safe}}$ has a finite *bad prefix* $\hat{\sigma} \in (2^{AP})^*$ that cannot be extended to any word in P_{safe}.
- **Liveness (P_{live}):** Any finite prefix $\hat{\sigma} \in (2^{AP})^*$ can be extended to an infinite word in P_{live}.

System Models, Paths and Traces

Example: Safety and Liveness on $K_{example}$

- **Safety:** *"Output o never turns true unless input l is true."*
  → In this case, $\hat{\sigma} = \varnothing\ \{o\}$ is a **bad prefix**. It has already violated the safety rule.
- **Liveness:** *"Output o will eventually turn true."*
  → Even if we observe $\varnothing\ \{l\}\ \{l\}\ \{l\}$ ... for 1000 steps, it is **not a bad prefix**. o might still turn true at step 1001.

System Models, Paths and Traces

Overcoming limitations with Hyperproperties

Because $K \vDash P \iff \forall \sigma \in \mathit{Traces}(K).\, \sigma \in P$, a trace property can **only** judge whether an individual execution σ is legal in isolation.

$\Rightarrow$ **It cannot compare two distinct executions σ and σ' of K against each other!**

Noninterference [1]

Information-Flow Security & Noninterference

- Consider a module with a **secret input** h (high security), **public input** l (low security), and a **public output** o.
- **Noninterference:** Output o must leak *no information* about secret h. Whenever two executions receive identical low inputs l at all times, they must produce identical public outputs o at all times.

Noninterference

Example: Is Trace σ a Bug?

- **Design A:** Sets $o := l$ (ignores secret input h).
- **Design B – $K_{example}$:** Sets $o := h$ (leaking secret input h).
- Suppose we observe **one single trace** from $K_{example}$:

$$\sigma = \varnothing \ \{h, l, o\} \ \{h, l, o\} \ \ldots$$

- σ is **valid** for both designs. But we can't tell if it violates noninterference or not.
  → Noninterference requires comparing two different traces!

Noninterference

Example: Comparing Two Traces Side-by-Side

We only catch the leak in Design B ($K_{example}$) when we compare σ (path $p = s_0 s_1 s_1 \ldots$) **simultaneously** with a second trace σ' (path $p' = s_0 s_2 s_2 \ldots$) where l is still `true`, but secret h is **not**.

Execution	**Step 0**	**Step 1**	**Step 2**	**Observation**
Trace σ (p)	$\varnothing$	$\{h, l, o\}$	$\{h, l, o\}$	Public input l is `true`, output o is `true`
Trace σ' (p')	$\varnothing$	$\{l\}$	$\{l\}$	Public input l is `true`, output o is `false`

- Looking at (σ, σ') together reveals that changing secret h changed public output o!
- **Why CTL / CTL* also fail:** Path quantifiers $\forall\varphi$ and $\exists\varphi$ in CTL* only allow φ to refer implicitly to *the single path* just quantified – never to two paths π and π' at once.

Outline

1. Basic Definitions for Model Checking and Motivation for Hyperproperties
2. ω-Regular Languages and LTL Model Checking
   - ω-Regular Languages and Büchi Automata
   - Model Checking ω - Regular Properties
   - LTL Syntax, Semantics and Normal Form
   - Automata-Based LTL Model Checking
3. HyperLTL and HyperCTL*
4. Automata-Theoretic Approach for checking HyperCTL*
5. Checking Circuits and Industrial Case Studies

ω-Regular Languages and Büchi Automata [2]

ω-Regular Languages

An **ω-regular expression** G over alphabet Σ has the form:

$$G = E_1 \cdot F_1^\omega + \cdots + E_n \cdot F_n^\omega$$

where E_i, F_i are regular expressions over Σ and $\varepsilon \notin \mathcal{L}(F_i)$. Infinite repetition $L^\omega = \{w_1 w_2 w_3 \ldots \mid w_k \in L\}$.

ω-Regular Languages and Büchi Automata

Example: ω-Regular Expressions over $\Sigma = 2^{\{a,b\}}$

Let a and $\neg a$ denote the sets $\{A \subseteq \{a,b\} \mid a \in A\}$ and $\{A \subseteq \{a,b\} \mid a \notin A\}$:

- **"Infinitely often a":** $\big((\neg a)^* \cdot a\big)^\omega$
  - Accepts $\{b\}\{a\}\{b\}\{b\}\{a\}\ldots$, rejects $\{a\}\{a\}\{b\}\{b\}\{b\}\ldots$
- **"Eventually forever a":** $\text{true}^* \cdot a^\omega$
  - Accepts $\{b\}\{a\}\{b\}\{a\}\{a\}\{a\}\ldots$, rejects $(\{a\}\{b\})^\omega$
- **"a holds infinitely often, or b never holds":** $\big((\neg a)^* \cdot a\big)^\omega + (\neg b)^\omega$

ω-Regular Languages and Büchi Automata

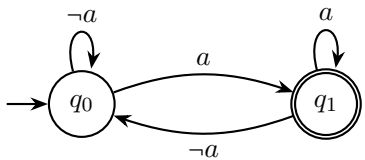
Nondeterministic Büchi Automata

Büchi Automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$.

- A **run** for $\sigma = A_0 A_1 A_2 \cdots \in \Sigma^\omega$ is an infinite sequence $q_0 q_1 q_2 \cdots \in Q^\omega$ of States with $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, A_i)$.
- **Büchi Acceptance:** Run $q_0 q_1 \ldots$ is accepting iff it visits F infinitely often.

Example 1

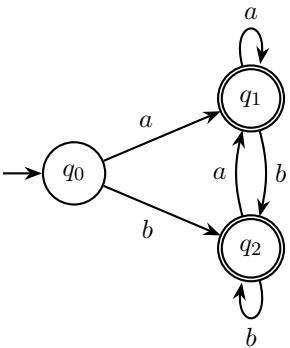
Consider $\mathcal{A} = (\{q_0, q_1\}, 2^{\{a,b\}}, \delta, \{q_0\}, \{q_1\})$ accepting $\mathcal{L}_\omega(\mathcal{A}) = \left((\neg a)^* a\right)^\omega$:



ω-Regular Languages and Büchi Automata

Example 2

Consider $\mathcal{A} = (Q, \Sigma, \delta, \{q_0\}, F)$ over $\Sigma = \{a, b\}$ with $Q = \{q_0, q_1, q_2\}$ and $F = \{q_1, q_2\}$:



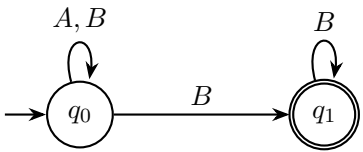
ω-Regular Languages and Büchi Automata

Key Properties of NBAs

1. **Expressiveness:** The class of languages accepted by NBAs is exactly the ω-regular languages (closed under $\cup, \cap$, and complement).
2. **NBA** $>$ **DBA:** Deterministic Büchi Automata (DBAs) are *strictly less expressive* than NBAs.

Example: Why NBA > DBA for $L = (A + B)^* B^\omega$

$L = (A + B)^* B^\omega$ over $\Sigma = \{A, B\}$:

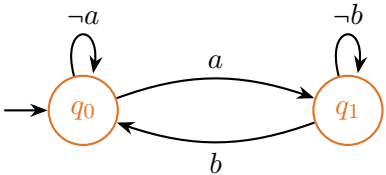


Generalized NBA

- Uses a set of acceptance sets $\mathcal{F} = \{F_1, \ldots, F_k\} \subseteq 2^Q$. A run is accepting iff it visits *every* $F_j \in \mathcal{F}$ infinitely often.
- **GNBA to NBA**: Any GNBA $\mathcal{G}$ with $k = |\mathcal{F}|$ acceptance sets can be converted into an equivalent NBA $\mathcal{A}$ with $|Q| \cdot k$ states (k copies $Q \times \{1, \ldots, k\}$ cycling $i \to i + 1$ upon visiting F_i, with accepting set $F_1 \times \{1\}$).

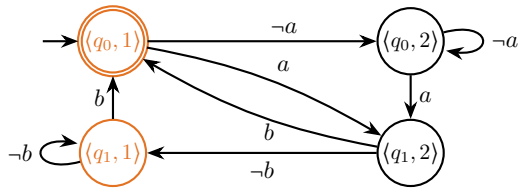
Example: GNBA to NBA

GNBA $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ with $\Sigma = 2^{\{a,b\}}$, $Q = \{q_0, q_1\}$, $Q_0 = \{q_0\}$, and $\mathcal{F} = \{F_1, F_2\}$ where $F_1 = \{q_0\}$ and $F_2 = \{q_1\}$:



Example: GNBA to NBA

Equivalent NBA $\mathcal{A} = (Q', \Sigma, \delta', Q_0', F')$ constructed via $Q' = Q \times \{1, 2\}$ and $F' = F_1 \times \{1\} = \{\langle q_0, 1\rangle\}$:



Model Checking ω - Regular Properties [2]

Reduction to Persistence Checking

- Given finite TS and ω-regular property P, let NBA $\mathcal{A}_{\overline{P}}$ accept the **bad traces** $\overline{P} = (2^{AP})^{\omega} \setminus P$
- Then $TS \models P \iff Traces(TS) \cap \mathcal{L}_{\omega}(\mathcal{A}_{\overline{P}}) = \varnothing$.
- Build synchronous product $TS \otimes \mathcal{A}_{\overline{P}}$ with states $\langle s, q \rangle$.
- $TS \models P$ iff $TS \otimes \mathcal{A}_{\overline{P}}$ satisfies the persistence property “eventually forever no accept state in F”.

Model Checking ω - Regular Properties

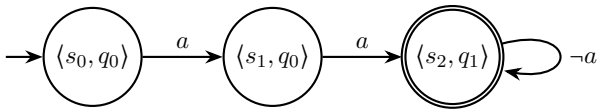
We just need to find Cycles

- In a finite graph, $TS \otimes \mathcal{A}_{\overline{P}} \not\models$ "eventually forever no accept state in F" iff there exists a **reachable accepting state** $\langle s, q \rangle$ **(**$q \in F$**) that lies on a cycle**.
- Finding such a cycle yields a **lasso-shaped counterexample** $\langle s_0, q_1 \rangle \dots \langle s, q \rangle (\dots \langle s, q \rangle)^\omega$.

Model Checking ω-Regular Properties

Example: Lasso Counterexample in $TS \otimes \mathcal{A}_{\neg P}$

We verify safety property P = “always a“. Its negation is ¬ P = “eventually $\neg a$“.



- **Lasso Counterexample Trace:** $\underbrace{\{a\}\{a\}}_{\text{Prefix}} \big(\underbrace{\{b\}}_{\text{Lasso cycle}} \big)^{\omega}$
- **Conclusion:** $TS \not\models \mathbf{G}\, a$.

Model Checking ω - Regular Properties

Nested Depth-First Search

Naive cycle check from each $q \in F$ takes $\mathcal{O}(N^2)$ time.
Nested DFS runs in **linear time** $\mathcal{O}(N+M)$ by interleaving two DFS traversals with shared visited sets R (outer) and T (inner):

1. **Outer DFS (`reachable_cycle`):** Explores reachable states using stack U.
2. **Post-Order Trigger:** Only when an accepting state $s' \in F$ is *fully expanded* in the outer DFS ($Post(s') \subseteq R$, about to pop from U), invoke **Inner DFS (`cycle_check`(s'))**.
3. **Inner DFS:** Searches from s' (ignoring states already in T) for a backward edge to s' (or to outer stack U).

LTL Syntax, Semantics and Normal Form [2]

LTL Syntax

$$\varphi ::= \text{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2$$

- **Finally:** $\mathbf{F}\,\varphi \stackrel{\text{def}}{=} \text{true}\ \mathbf{U}\ \varphi$
- **Generally:** $\mathbf{G}\,\varphi \stackrel{\text{def}}{=} \neg\mathbf{F}\,\neg\varphi$
- **Weak Until:** $\varphi_1\ \mathbf{W}\ \varphi_2 \stackrel{\text{def}}{=} (\varphi_1\ \mathbf{U}\ \varphi_2) \vee \mathbf{G}\,\varphi_1$
- **Release:** $\varphi_1\ \mathbf{R}\ \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1\ \mathbf{U}\ \neg\varphi_2)$

LTL Syntax, Semantics and Normal Form

Semantics over Words $\sigma = A_0 A_1 \cdots \in (2^{AP})^\omega$

- $\sigma \vDash a \iff a \in A_0$
- $\sigma \vDash \mathbf{X}\,\varphi \iff \sigma[1..] = A_1 A_2 \cdots \vDash \varphi$
- $\sigma \vDash \varphi_1\ \mathbf{U}\ \varphi_2 \iff \exists j \geq 0.\, \sigma[j..] \vDash \varphi_2 \wedge \forall 0 \leq i < j.\, \sigma[i..] \vDash \varphi_1$
- **System Satisfaction:** $TS \vDash \varphi \iff \mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi)$.
- **Caution on Negation:** $\pi \nvDash \varphi \iff \pi \vDash \neg\varphi$ holds on single paths, but $TS \nvDash \varphi \not\Longleftrightarrow TS \vDash \neg\varphi$.

LTL Syntax, Semantics and Normal Form

Example: Specifying Properties in LTL

Pattern	LTL Formula	Intuition
Safety (Mutex)	$\mathbf{G}\,\neg(crit_1 \wedge crit_2)$	Never both in critical section simultaneously
Liveness	$\mathbf{G}\,\mathbf{F}\,crit_1$	Process 1 enters critical section infinitely often
Persistence	$\mathbf{F}\,\mathbf{G}\,stable$	Eventually remains $stable$ forever
Response	$\mathbf{G}\,(req \Rightarrow \mathbf{F}\,ack)$	Every request req is eventually answered by ack

LTL Syntax, Semantics and Normal Form

Expansion Laws and Positive Normal Form

- **Expansion Laws** split temporal operators into a *present* check and a *next-step* check:

$$\varphi_1 \,\mathbf{U}\, \varphi_2 \equiv \varphi_2 \vee \big(\varphi_1 \wedge \mathbf{X}\,(\varphi_1 \,\mathbf{U}\, \varphi_2)\big)$$
$$\varphi_1 \,\mathbf{W}\, \varphi_2 \equiv \varphi_2 \vee \big(\varphi_1 \wedge \mathbf{X}\,(\varphi_1 \,\mathbf{W}\, \varphi_2)\big)$$
$$\varphi_1 \,\mathbf{R}\, \varphi_2 \equiv \varphi_2 \wedge \big(\varphi_1 \vee \mathbf{X}\,(\varphi_1 \,\mathbf{R}\, \varphi_2)\big)$$

- **Release PNF:** Pushing $\neg$ inward using $\neg\mathbf{X}\,\varphi \equiv \mathbf{X}\,\neg\varphi$ and $\neg(\varphi_1 \,\mathbf{U}\, \varphi_2) \equiv \neg\varphi_1 \,\mathbf{R}\, \neg\varphi_2$ puts any φ into PNF with **linear blowup** $\mathcal{O}(|\varphi|)$.

Quiz: Why would introducing $\mathbf{W}$ instead of $\mathbf{R}$ cause exponential blowup?

Question

Negations are being pushed inside a $\mathbf{U}$-formula by introducing $\mathbf{R}$. Why would introducing $\mathbf{W}$ instead of $\mathbf{R}$ cause exponential blowup?

Quiz: Why would introducing $\mathbf{W}$ instead of $\mathbf{R}$ cause exponential blowup?

Question

Negations are being pushed inside a $\mathbf{U}$-formula by introducing $\mathbf{R}$. Why would introducing $\mathbf{W}$ instead of $\mathbf{R}$ cause exponential blowup?

Solution

Because $\mathbf{W}$ requires duplicating subformulas when expanding negations, whereas $\mathbf{R}$ maintains strict structural sharing ($\mathcal{O}(|\varphi|)$) without subformula duplication.
$\neg(\varphi_1 \mathbf{U} \varphi_2) \equiv (\varphi_1 \wedge \neg\varphi_2) \mathbf{W} (\neg\varphi_1 \wedge \neg\varphi_2)$

Automata-Based LTL Model Checking [2]

LTL → GNBA Construction

- Given LTL formula φ, define $closure(\varphi) = subf(\varphi) \cup \{\neg\psi \mid \psi \in subf(\varphi)\}$.
- **States Q:** All *elementary* sets $B \subseteq closure(\varphi)$ that are:
  1. *Propositionally consistent* ($\psi_1 \wedge \psi_2 \in B \iff \psi_1 \in B \wedge \psi_2 \in B$; $\psi \in B \Rightarrow \neg\psi \notin B$),
  2. *Maximal* ($\psi \notin B \Rightarrow \neg\psi \in B$),
  3. *Locally* $\mathbf{U}$*-consistent* ($\varphi_2 \in B \Rightarrow \varphi_1 \mathbf{U} \varphi_2 \in B$; and $\varphi_1 \mathbf{U} \varphi_2 \in B \wedge \varphi_2 \notin B \Rightarrow \varphi_1 \in B$).
- **Initial States:** $Q_0 = \{B \in Q \mid \varphi \in B\}$.
- **Transitions** $B' \in \delta(B, A)$**:** Requires $A = B \cap AP$ and:
  - $\forall \mathbf{X}\ \psi \in closure(\varphi) :\ \mathbf{X}\ \psi \in B \iff \psi \in B'$
  - $\forall \varphi_1\ \mathbf{U}\ \varphi_2 \in closure(\varphi) :\ \varphi_1\ \mathbf{U}\ \varphi_2 \in B \iff \big(\varphi_2 \in B \vee (\varphi_1 \in B \wedge\ \varphi_1\ \mathbf{U}\ \varphi_2 \in B')\big)$

LTL → GNBA Construction

- **Acceptance Sets $\mathcal{F}$:** Add one acceptance set per Until subformula:

$$F_{\varphi_1 \,\mathbf{U}\, \varphi_2} = \{B \in Q \mid \varphi_1 \mathbf{U} \varphi_2 \notin B \text{ or } \varphi_2 \in B\}$$

Automata-Based LTL Model Checking

Example: $\varphi = a \ \mathbf{U} \ b$

1. Closure: $closure(\varphi) = \{a, \neg a, b, \neg b, a\mathbf{U}b, \neg(a\mathbf{U}b)\}$.

2. Elementary States:

- $B_1 = \{a,\ b,\ a\mathbf{U}b\}$
- $B_2 = \{\neg a,\ b,\ a\mathbf{U}b\}$
- $B_3 = \{a,\ \neg b,\ a\mathbf{U}b\}$
- $B_4 = \{a,\ \neg b,\ \neg(a\mathbf{U}b)\}$
- $B_5 = \{\neg a,\ \neg b,\ \neg(a\mathbf{U}b)\}$

3. Initial States: $\{B_1, B_2, B_3\}$

4. Acceptance Set: $\{\{B_1, B_2, B_4, B_5\}\}$

Quiz: Why Use Multiple Acceptance Sets in GNBA?

Question

Why does the LTL-to-GNBA construction introduce multiple acceptance sets $F_{\varphi_1 \mathbf{U} \varphi_2}$ (one per Until subformula) rather than a single Büchi acceptance set F?

Quiz: Why Use Multiple Acceptance Sets in GNBA?

Question

Why does the LTL-to-GNBA construction introduce multiple acceptance sets $F_{\varphi_1 \mathbf{U} \varphi_2}$ (one per Until subformula) rather than a single Büchi acceptance set F?

Solution

Because different Until formulas $\varphi_1 \mathbf{U} \varphi_2$ have independent fulfillment obligations. A GNBA with multiple acceptance sets decouples these checks so that every Until fulfillment obligation is visited infinitely often.

Automata-Based LTL Model Checking

Overall LTL Model-Checking Pipeline and Complexity

1. Negate formula φ and construct GNBA $\mathcal{G}_{\neg\varphi}$ of size $2^{\mathcal{O}(|\varphi|)}$ with $\leq |\varphi|$ acceptance sets.
2. Convert $\mathcal{G}_{\neg\varphi}$ into NBA $\mathcal{A}_{\neg\varphi}$ with $\mathcal{L}_\omega(\mathcal{A}_{\neg\varphi}) = Words(\neg\varphi)$ (size $2^{\mathcal{O}(|\varphi|)}$).
3. Build product $TS \otimes \mathcal{A}_{\neg\varphi}$ and run Nested DFS to check $TS \otimes \mathcal{A}_{\neg\varphi} \vDash \mathbf{F}\ \mathbf{G}\ \neg F$.

Automata-Based LTL Model Checking

Complexity of LTL Model Checking

- **Time Complexity:** $\mathcal{O}\big(|TS| \cdot 2^{\mathcal{O}(|\varphi|)}\big)$.
- **Space / Decision Complexity:**
  - **PSPACE-complete** in $|\varphi|$
  - **NLOGSPACE-complete** in $|TS|$ (guessing a lasso on the fly)

Outline

1 Basic Definitions for Model Checking and Motivation for Hyperproperties

2 ω-Regular Languages and LTL Model Checking

3 HyperLTL and HyperCTL*
- Syntax
- Semantics

4 Automata-Theoretic Approach for checking HyperCTL*

5 Checking Circuits and Industrial Case Studies

6 Questions?

Syntax [1]

Syntax for HyperLTL and HyperCTL*

Grammar for Formulas in HyperLTL and HyperCTL*:

$$\varphi ::= \text{true} \mid a_\pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \varphi\ \mathbf{U}\ \varphi \mid \varphi\ \mathbf{R}\ \varphi \mid \exists\pi.\,\varphi \mid \forall\pi.\,\varphi$$

- Atomic propositions a_π are indexed by a path variable
- We require that Temporal operators only occur inside the scope of path quantifiers
- **HyperLTL (Linear-Time):** Sublogic of formulas in **prenex normal form:** Quantifiers only at start of the formula
- **HyperCTL* (Branching-Time):** Quantifiers may appear inside temporal operators; new paths start at the current state of the *most recently quantified path* ϵ.

Syntax

Example: HyperLTL and HyperCTL* Interpretation

1. **Existence of a Secret Twin (HyperLTL):**

$$\forall \pi_1 . \exists \pi_2 . \mathbf{X}(h_{\pi_1} \neq h_{\pi_2}) \wedge \mathbf{G}(o_{\pi_1} = o_{\pi_2})$$

Meaning: "For every path π_1, there is another path π_2 with the *opposite* secret input h at the next step, yet producing the exact same output o forever."

2. **Next Branching Output (HyperCTL*):**

$$\forall \pi_1 . \mathbf{G}\Big(l_{\pi_1} \Rightarrow \exists \pi_2 . \mathbf{X}(\neg h_{\pi_2} \wedge (o_{\pi_1} = o_{\pi_2})\Big)$$

Meaning: "Along path π_1, whenever l is true at any step t, from *that exact state at step* t there branches a path π_2 that produces the same next o, without h in the next step."

Syntax

Example: Noninterference in HyperLTL

Noninterference in HyperLTL (Relating Two Named Paths π and π'):

$$\forall \pi.\, \forall \pi'.\, \mathbf{G} \left(\bigwedge_{i \in I \setminus \{h\}} i_\pi = i_{\pi'} \right) \Rightarrow \mathbf{G}\, (o_\pi = o_{\pi'})$$

- Here a_π means "atomic proposition a holds on path π", and $x_\pi = x_{\pi'}$ abbreviates $(x_\pi \leftrightarrow x_{\pi'})$.
- **Other Examples:** Symmetric access in mutual exclusion protocols and minimum Hamming distance in error-correcting codes are also hyperproperties!

Semantics [1]

Semantics for HyperLTL and HyperCTL*

Let K be a Kripke structure.

Semantics is given in terms of *assignments* $\Pi : N \to \mathit{Paths}^*(K, s_0)$ of a set of path variables N to suffixes of paths:

$\Pi \vDash_K a_\pi$ iff $a \in L(\Pi(\pi)(0))$

$\Pi \vDash_K \neg\varphi$ iff $\Pi \nvDash_K \varphi$

$\Pi \vDash_K \varphi_1 \vee \varphi_2$ iff $\Pi \vDash_K \varphi_1$ or $\Pi \vDash_K \varphi_2$

$\Pi \vDash_K \mathbf{X}\, \varphi$ iff $\Pi[1, \infty] \vDash_K \varphi$

$\Pi \vDash_K \varphi_1\, \mathbf{U}\, \varphi_2$ iff $\exists i \geq 0 : \Pi[i, \infty] \vDash_K \varphi_2 \wedge \forall 0 \leq j < i : \Pi[j, \infty] \vDash_K \varphi_1$

$\Pi \vDash_K \exists \pi. \varphi$ iff $\exists p \in \mathit{Paths}(K, \Pi(\epsilon)(0)) : \Pi[\pi \mapsto p, \epsilon \mapsto p] \vDash_K \varphi$

For the empty assignment $\Pi = \{\}$, we define $\Pi(\varepsilon)(0)$ to yield initial state of K

Intuitively: $K \vDash \varphi \iff \{\} \vDash_K \varphi$.

Semantics

Example: How Π and $\Pi[i, \infty]$ works on $K_{example}$

We assign $\pi_1 \mapsto s_0 s_1 s_1 \ldots$ and $\pi_2 \mapsto s_0 s_2 s_2 \ldots$ in our table Π:

Path Variable in Π	**Step 0**	**Step 1**	**Step 2** $\ldots$
$\Pi(\pi_1)$	$s_0 : \varnothing$	$s_1 : \{h, l, o\}$	$s_1 : \{h, l, o\} \ldots$
$\Pi(\pi_2)$	$s_0 : \varnothing$	$s_2 : \{l\}$	$s_2 : \{l\} \ldots$

- **Evaluating Atomic Propositions at Step 0:**
  - $\Pi \vDash_K (o_{\pi_1} = o_{\pi_2})$ holds right now. At Step 0: $o \notin L(s_0)$ on π_1 and $o \notin L(s_0)$ on π_2, so both are `false`.

Semantics

Example: How Π and $\Pi[i, \infty]$ works on $K_{example}$

We assign $\pi_1 \mapsto s_0 s_1 s_1 \ldots$ and $\pi_2 \mapsto s_0 s_2 s_2 \ldots$ in our table Π:

Path Variable in Π	**Step 0**	**Step 1**	**Step 2** $\ldots$
$\Pi(\pi_1)$	$s_0 : \varnothing$	$s_1 : \{h, l, o\}$	$s_1 : \{h, l, o\} \ldots$
$\Pi(\pi_2)$	$s_0 : \varnothing$	$s_2 : \{l\}$	$s_2 : \{l\} \ldots$

- **Evaluating $\Pi \vDash_K \mathbf{X}(o_{\pi_1} = o_{\pi_2})$ via Lockstep Shift $\Pi[1, \infty]$:**
  - $\Pi[1, \infty]$ **chops off Step 0 from ALL paths simultaneously**, so Step 1 becomes the new Step 0:

$$\Pi[1, \infty](\pi_1)(0) = s_1\ (\{h, l, o\}) \qquad \text{and} \qquad \Pi[1, \infty](\pi_2)(0) = s_2\ (\{l\})$$

  - Now $o \in L(s_1)$ is `true`, but $o \in L(s_2)$ is `false` $\implies \Pi[1, \infty] \not\vDash_K (o_{\pi_1} = o_{\pi_2})$!

Quiz: LTL and HyperLTL Formalization

Question

Over $AP = \{h, l, o\}$, decide if LTL is enough or HyperLTL is required, and write the formula:

1. *"Whenever l holds, o holds now or in the next step."*
2. *"Any two runs with identical l at all steps have identical o at all steps."*
3. *"There exist two runs with opposite h at the next step, identical l forever, and identical o forever."*

Quiz: LTL and HyperLTL Formalization

Question

Over $AP = \{h, l, o\}$, decide if LTL is enough or HyperLTL is required, and write the formula:

1. *"Whenever l holds, o holds now or in the next step."*
2. *"Any two runs with identical l at all steps have identical o at all steps."*
3. *"There exist two runs with opposite h at the next step, identical l forever, and identical o forever."*

Solution

1. LTL: $\mathbf{G}\ (l \Rightarrow (o \vee \mathbf{X}\ o))$
2. HyperLTL: $\forall \pi.\ \forall \pi'.\ \mathbf{G}\ (l_\pi = l_{\pi'}) \Rightarrow \mathbf{G}\ (o_\pi = o_{\pi'})$
3. HyperLTL: $\exists \pi.\ \exists \pi'.\ \mathbf{X}\ (h_\pi \neq h_{\pi'}) \wedge \mathbf{G}\ (l_\pi = l_{\pi'} \wedge o_\pi = o_{\pi'})$

Outline

1 Basic Definitions for Model Checking and Motivation for Hyperproperties

2 ω-Regular Languages and LTL Model Checking

3 HyperLTL and HyperCTL*

4 Automata-Theoretic Approach for checking HyperCTL*
- ABA and K-Equivalence
- Constructing Automata for HyperCTL* formulas
- Complexity and full HyperCTL*

5 Checking Circuits and Industrial Case Studies

ABA and K-Equivalence [1]

Alternating Büchi Automata (ABA)

An ABA is a tuple $\mathcal{A} = (Q, q_0, \Sigma, \rho, F)$
Like NBA, but the transition function $\rho : Q \times \Sigma \to \mathbb{B}^+(Q)$ maps a state and letter to a **positive Boolean formula** over Q (using both $\vee$ and $\wedge$).

- A run of a word $\pi \in \Sigma^\omega$ is a ***Q*-Labeled tree** (T, r) with root $r(\epsilon) = q_0$.
- At node τ at depth $i = |\tau|$ with state $q = r(\tau)$ and children $\tau_1, \ldots, \tau_k$:

$$\{r(\tau_1), \ldots, r(\tau_k)\} \vDash \rho(q, \pi[i])$$

Acceptance: Every infinite branch $\tau_0\tau_1\ldots$ in T visits F infinitely often.
Intuition: For an accepting run we need to pick those sets of states that fulfil the boolean formula and lead to accepting paths down the Automaton.

Small ABA example

Suppose we want to find an accepting run for $(ab)^\omega$ on the ABA
$\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, q_0, \{a, b\}, \rho, \{q_1, q_3\})$
with ρ from the table:

$\rho(q_0, a)$	$q_1 \wedge q_2$
$\rho(q_0, b)$	$q_3 \vee q_2$
$\rho(q_1, a)$	$q_2 \wedge q_3$
$\rho(q_1, b)$	$q_0 \wedge q_3$
$\rho(q_2, a)$	$q_2 \wedge q_1$
$\rho(q_2, b)$	$q_3 \vee q_0$
$\rho(q_3, a)$	$q_1 \wedge q_2$
$\rho(q_3, b)$	$q_1 \wedge q_2$

Small ABA example

Suppose we want to find an accepting run for $(ab)^\omega$ on the ABA
$\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, q_0, \{a, b\}, \rho, \{q_1, q_3\})$
with ρ from the table:

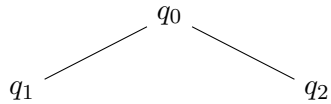
$\rho(q_0, a)$	$q_1 \wedge q_2$
$\rho(q_0, b)$	$q_3 \vee q_2$
$\rho(q_1, a)$	$q_2 \wedge q_3$
$\rho(q_1, b)$	$q_0 \wedge q_3$
$\rho(q_2, a)$	$q_2 \wedge q_1$
$\rho(q_2, b)$	$q_3 \vee q_0$
$\rho(q_3, a)$	$q_1 \wedge q_2$
$\rho(q_3, b)$	$q_1 \wedge q_2$

q_0

Small ABA example

Suppose we want to find an accepting run for $(ab)^\omega$ on the ABA
$\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, q_0, \{a, b\}, \rho, \{q_1, q_3\})$
with ρ from the table:

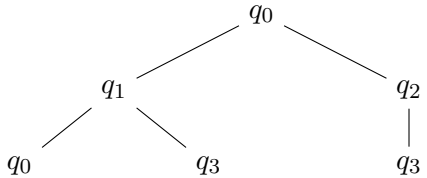
$\rho(q_0, a)$	$q_1 \wedge q_2$
$\rho(q_0, b)$	$q_3 \vee q_2$
$\rho(q_1, a)$	$q_2 \wedge q_3$
$\rho(q_1, b)$	$q_0 \wedge q_3$
$\rho(q_2, a)$	$q_2 \wedge q_1$
$\rho(q_2, b)$	$q_3 \vee q_0$
$\rho(q_3, a)$	$q_1 \wedge q_2$
$\rho(q_3, b)$	$q_1 \wedge q_2$



Small ABA example

Suppose we want to find an accepting run for $(ab)^\omega$ on the ABA
$\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, q_0, \{a, b\}, \rho, \{q_1, q_3\})$
with ρ from the table:

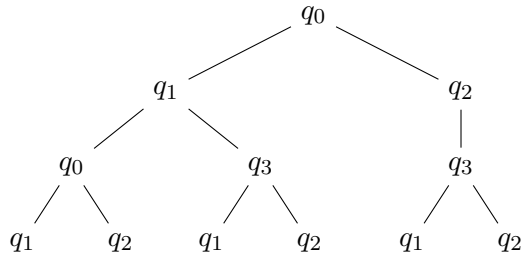
$\rho(q_0, a)$	$q_1 \wedge q_2$
$\rho(q_0, b)$	$q_3 \vee q_2$
$\rho(q_1, a)$	$q_2 \wedge q_3$
$\rho(q_1, b)$	$q_0 \wedge q_3$
$\rho(q_2, a)$	$q_2 \wedge q_1$
$\rho(q_2, b)$	$q_3 \vee q_0$
$\rho(q_3, a)$	$q_1 \wedge q_2$
$\rho(q_3, b)$	$q_1 \wedge q_2$



Small ABA example

Suppose we want to find an accepting run for $(ab)^\omega$ on the ABA
$\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, q_0, \{a, b\}, \rho, \{q_1, q_3\})$
with ρ from the table:

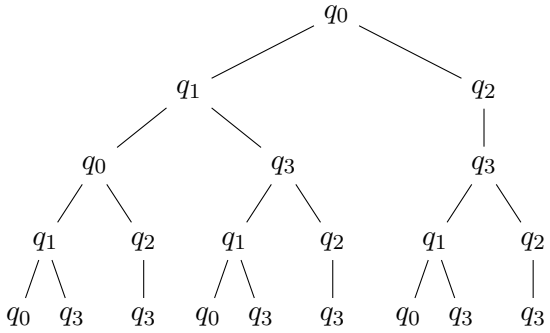
$\rho(q_0, a)$	$q_1 \wedge q_2$
$\rho(q_0, b)$	$q_3 \vee q_2$
$\rho(q_1, a)$	$q_2 \wedge q_3$
$\rho(q_1, b)$	$q_0 \wedge q_3$
$\rho(q_2, a)$	$q_2 \wedge q_1$
$\rho(q_2, b)$	$q_3 \vee q_0$
$\rho(q_3, a)$	$q_1 \wedge q_2$
$\rho(q_3, b)$	$q_1 \wedge q_2$



Small ABA example

Suppose we want to find an accepting run for $(ab)^\omega$ on the ABA
$\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, q_0, \{a, b\}, \rho, \{q_1, q_3\})$
with ρ from the table:

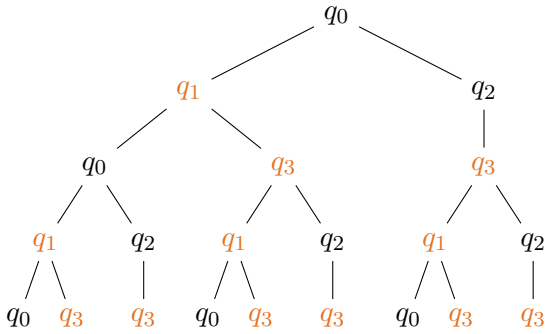
$\rho(q_0, a)$	$q_1 \wedge q_2$
$\rho(q_0, b)$	$q_3 \vee q_2$
$\rho(q_1, a)$	$q_2 \wedge q_3$
$\rho(q_1, b)$	$q_0 \wedge q_3$
$\rho(q_2, a)$	$q_2 \wedge q_1$
$\rho(q_2, b)$	$q_3 \vee q_0$
$\rho(q_3, a)$	$q_1 \wedge q_2$
$\rho(q_3, b)$	$q_1 \wedge q_2$



Small ABA example

Suppose we want to find an accepting run for $(ab)^\omega$ on the ABA
$\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, q_0, \{a, b\}, \rho, \{q_1, q_3\})$
with ρ from the table:

$\rho(q_0, a)$	$q_1 \wedge q_2$
$\rho(q_0, b)$	$q_3 \vee q_2$
$\rho(q_1, a)$	$q_2 \wedge q_3$
$\rho(q_1, b)$	$q_0 \wedge q_3$
$\rho(q_2, a)$	$q_2 \wedge q_1$
$\rho(q_2, b)$	$q_3 \vee q_0$
$\rho(q_3, a)$	$q_1 \wedge q_2$
$\rho(q_3, b)$	$q_1 \wedge q_2$



ABA and K-Equivalence

Converting ABAs to NBAs [3]

Miyano & Hayashi algorithm:
For any ABA $\mathcal{A}$ with n states, there is a NBA $MH(\mathcal{A})$ with $2^{\mathcal{O}(n)}$ states that accepts the exact same language $\mathcal{L}_\omega(\mathcal{A})$.
Basic idea: Our NBA simulates the tree of a run level by level and keeps track of which nodes have a path to the last "reset point" that contains a final state and which nodes dont. When all nodes do, we "reset" and do the same again.
Being able to hit reset points infinitely often means, that the word has a tree that fulfills the ABA acceptance condition.
$\Longrightarrow$ reset points are final states

ABA and K-Equivalence

Changes to the alphabet

Goal: Build Automaton $\mathcal{A}_\varphi$ for HyperCTL* formula bottom-up from the syntax tree of φ
Problem: we have subformulas with multiple path variables
Solution: have the automata $\mathcal{A}_\psi$ for subformulas ψ work on words that are tuples of States:
Let ψ be a subformula with n path variables and $K = (S, s_0, \delta, \mathsf{AP}, L)$ a Kripke Structure:

- Alphabet of corresponding automaton $\mathcal{A}_\psi$: $\Sigma_\psi = S^n$
- **K-Equivalence:** $\mathcal{A}_\psi$ is K-equivalent to $\psi \iff$ its accepted words $(s_1^0, \ldots, s_n^0)(s_1^1, \ldots, s_n^1) \cdots \in \mathcal{L}_\omega(\mathcal{A}_\psi)$ correspond exactly to the path assignments $\Pi(\pi_i) = s_i^0 s_i^1 \ldots$ such that $\Pi \vDash_K \psi$.

Constructing Automata for HyperCTL* formulas [1]

Alternation-Free HyperCTL*

A formula in NNF is *alternation-free* if it contains quantifiers of only one type, and no $\exists$ occurs in the left subformula of $\mathbf{U}$ or right of $\mathbf{R}$ (symmetrically, no $\forall$ in the right of $\mathbf{U}$ or left of $\mathbf{R}$).

Construction for Alternation-Free HyperCTL*

$\psi = a_{\pi_k}$	$\mathcal{A}_\psi = (\{q_0\}, q_0, \Sigma, \rho, \varnothing)$, where $\rho(q_0, \mathbf{s}) = (a \in L(s\|_k))$
$\psi = \neg a_{\pi_k}$	$\mathcal{A}_\psi = (\{q_0\}, q_0, \Sigma, \rho, \varnothing)$, where $\rho(q_0, \mathbf{s}) = (a \notin L(s\|_k))$
$\psi = \psi_1 \wedge \psi_2$	$\mathcal{A}_\psi = (Q_1 \cup Q_2 \cup \{q_0\}, q_0, \Sigma, \rho, F_1 \cup F_2)$ where $\rho(q_0, \mathbf{s}) = \rho_1(q_{0,1}, \mathbf{s}) \wedge \rho_2(q_{0,2}, \mathbf{s})$ and $\rho(q, \mathbf{s}) = \rho_i(q, \mathbf{s})$ for $q \in Q_i, i \in \{1, 2\}$
$\psi = \psi_1 \vee \psi_2$	$\mathcal{A}_\psi = (Q_1 \cup Q_2 \cup \{q_0\}, q_0, \Sigma, \rho, F_1 \cup F_2)$ where $\rho(q_0, \mathbf{s}) = \rho_1(q_{0,1}, \mathbf{s}) \vee \rho_2(q_{0,2}, \mathbf{s})$ and $\rho(q, \mathbf{s}) = \rho_i(q, \mathbf{s})$ for $q \in Q_i, i \in \{1, 2\}$

Construction for Alternation-Free HyperCTL*

$\psi = X\psi_1$	$\mathcal{A}_\psi = (Q_1 \cup \{q_0\}, q_0, \Sigma, \rho, F_1)$ where $\rho(q_0, \mathbf{s}) = q_{0,1}$ and $\rho(q, \mathbf{s}) = \rho_1(q, \mathbf{s})$ for $q \in Q_1$
$\psi = \psi_1 U \psi_2$	$\mathcal{A}_\psi = (Q_1 \cup Q_2 \cup \{q_0\}, q_0, \Sigma, \rho, F_1 \cup F_2)$ where $\rho(q_0, \mathbf{s}) = \rho_2(q_{0,2}, \mathbf{s}) \vee (\rho_1(q_{0,1}, \mathbf{s}) \wedge q_0)$ and $\rho(q, \mathbf{s}) = \rho_i(q, \mathbf{s})$ for $q \in Q_i, i \in \{1, 2\}$
$\psi = \psi_1 R \psi_2$	$\mathcal{A}_\psi = (Q_1 \cup Q_2 \cup \{q_0\}, q_0, \Sigma, \rho, F_1 \cup F_2 \cup \{q_0\})$ where $\rho(q_0, \mathbf{s}) = \rho_2(q_{0,2}, \mathbf{s}) \wedge (\rho_1(q_{0,1}, \mathbf{s}) \vee q_0)$ and $\rho(q, \mathbf{s}) = \rho_i(q, \mathbf{s})$ for $q \in Q_i, i \in \{1, 2\}$

Construction for Alternation-Free HyperCTL*

Handling $\exists \pi.\ \psi_1$ in our construction

To handle the existential quantifier we reduce the alphabet of the automaton $\mathcal{A}_{\psi_1}$ we constructed for ψ_1 like this:

1. Nondeterminize $\mathcal{A}_{\psi_1}$ into NBA $\mathcal{N}'_{\psi_1} = MH(\mathcal{A}_{\psi_1}) = (Q', q'_0, S^{n+1}, \rho', F')$
2. Take product with $K = (S, s_0, \delta, AP, L)$ to guess path p starting at $s|_n$ (or s_0 if $n = 0$): $\mathcal{A}_\psi = (Q' \times S \uplus \{q''_0\}, q''_0, S^n, \rho'', F' \times S)$ with:

$$\rho''(q''_0, \mathbf{s}) = \{(q', s') \mid q' \in \rho'(q'_0, \mathbf{s} + \mathbf{s}|_n),\ s' \in \delta(\mathbf{s}|_n)\}$$

and

$$\rho''((q, s), \mathbf{s}) = \{(q', s') \mid q' \in \rho'(q, \mathbf{s} + s),\ s' \in \delta(s)\}$$

Construction for Alternation-Free HyperCTL*

Formulas that dont start with $\exists\ \pi$

Since we defined for HyperCTL* that temporal operators only appear in the scope of path quantifiers, Alternation-Free formulas that dont start with $\exists\ \pi$ can be converted to a boolean combination over quantified subformulas.
Then we just use the construction for each subformula and check for emptiness.

Complexity and full HyperCTL* [1]

Complexity of checking alternation-free HyperCTL*

- Model checking alternation-free HyperCTL* is **PSPACE-complete in** $|\varphi|$ and **NLOGSPACE-complete in** $|K|$!
- **Proof Sketch:**
  - we can construct the NBA **during** the check $\implies$ we dont need to save full NBA

Complexity and full HyperCTL* [1]

Complexity of checking alternation-free HyperCTL*

- Model checking alternation-free HyperCTL* is **PSPACE-complete in** $|\varphi|$ and **NLOGSPACE-complete in** $|K|$!
- **Proof Sketch:**
  - we can construct the NBA **during** the check $\implies$ we dont need to save full NBA
  - we do **induction** over our construction to show that when we MH, we are still in PSPACE

Complexity and full HyperCTL* [1]

Complexity of checking alternation-free HyperCTL*

- Model checking alternation-free HyperCTL* is **PSPACE-complete in** $|\varphi|$ and **NLOGSPACE-complete in** $|K|$!
- **Proof Sketch:**
  - we can construct the NBA **during** the check $\implies$ we dont need to save full NBA
  - we do **induction** over our construction to show that when we MH, we are still in PSPACE
  - for $a_{\pi_k}, \neg a_{\pi_k}$ and $\psi_1 \vee \psi_2$ its trivial that the size of states is in PSPACE

Complexity and full HyperCTL* [1]

Complexity of checking alternation-free HyperCTL*

- Model checking alternation-free HyperCTL* is **PSPACE-complete in** $|\varphi|$ and **NLOGSPACE-complete in** $|K|$!
- **Proof Sketch:**
  - we can construct the NBA **during** the check $\implies$ we dont need to save full NBA
  - we do **induction** over our construction to show that when we MH, we are still in PSPACE
  - for $a_{\pi_k}, \neg a_{\pi_k}$ and $\psi_1 \vee \psi_2$ its trivial that the size of states is in PSPACE
  - $\psi_1 \wedge \psi_2$ doesnt add loops $\implies$ polynomial in the size of the corresp. automata

Complexity and full HyperCTL* [1]

Complexity of checking alternation-free HyperCTL*

- Model checking alternation-free HyperCTL* is **PSPACE-complete in** $|\varphi|$ and **NLOGSPACE-complete in** $|K|$!
- **Proof Sketch:**
  - we can construct the NBA **during** the check $\implies$ we dont need to save full NBA
  - we do **induction** over our construction to show that when we MH, we are still in PSPACE
  - for $a_{\pi_k}, \neg a_{\pi_k}$ and $\psi_1 \vee \psi_2$ its trivial that the size of states is in PSPACE
  - $\psi_1 \wedge \psi_2$ doesnt add loops $\implies$ polynomial in the size of the corresp. automata
  - $\mathbf{U}$ and $\mathbf{R}$ have self loops in conjunctions, but we defined alternation-free HyperCTL* in a way that the size of a state will be polynomial (no quant. where we do the conjunction)

Complexity and full HyperCTL* [1]

Complexity of checking alternation-free HyperCTL*

- Model checking alternation-free HyperCTL* is **PSPACE-complete in** $|\varphi|$ and **NLOGSPACE-complete in** $|K|$!
- **Proof Sketch:**
  - we can construct the NBA **during** the check $\implies$ we dont need to save full NBA
  - we do **induction** over our construction to show that when we MH, we are still in PSPACE
  - for $a_{\pi_k}, \neg a_{\pi_k}$ and $\psi_1 \vee \psi_2$ its trivial that the size of states is in PSPACE
  - $\psi_1 \wedge \psi_2$ doesnt add loops $\implies$ polynomial in the size of the corresp. automata
  - $\mathbf{U}$ and $\mathbf{R}$ have self loops in conjunctions, but we defined alternation-free HyperCTL* in a way that the size of a state will be polynomial (no quant. where we do the conjunction)
  - $\exists\ \pi.\ \psi_1$ should be able to use I.H.

Complexity and full HyperCTL* [1]

Complexity of checking alternation-free HyperCTL*

- Model checking alternation-free HyperCTL* is **PSPACE-complete in** $|\varphi|$ and **NLOGSPACE-complete in** $|K|$!
- **Proof Sketch:**
  - we can construct the NBA **during** the check $\implies$ we dont need to save full NBA
  - we do **induction** over our construction to show that when we MH, we are still in PSPACE
  - for $a_{\pi_k}, \neg a_{\pi_k}$ and $\psi_1 \vee \psi_2$ its trivial that the size of states is in PSPACE
  - $\psi_1 \wedge \psi_2$ doesnt add loops $\implies$ polynomial in the size of the corresp. automata
  - $\mathbf{U}$ and $\mathbf{R}$ have self loops in conjunctions, but we defined alternation-free HyperCTL* in a way that the size of a state will be polynomial (no quant. where we do the conjunction)
  - $\exists\ \pi.\ \psi_1$ should be able to use I.H.
  - Since NBA emptiness is in NLOGSPACE, checking $K \vDash \varphi$ runs in NLOGSPACE in $|K|$.

Complexity and full HyperCTL*

full HyperCTL* and Alternation Depth

- To handle $\psi = \neg\exists\pi.\,\psi_1$, we nondeterminize $\mathcal{N}'_{\psi_1} = MH(\mathcal{A}_{\psi_1})$, take the product with K, and **complement** the resulting automaton, which is in $\mathcal{O}(|Q|^2)$.
- **Alternation Depth k:** Highest number of switches $\exists \leftrightarrow \forall$ along any syntax tree path in NNF (plus $\exists$ in left of $\mathbf{U}$ / right of $\mathbf{R}$, or $\forall$ in right of $\mathbf{U}$ / left of $\mathbf{R}$).

Complexity and full HyperCTL*

complexity of full HyperCTL*

Let $g_c(0, n) = n$, $g_c(k, n) = c^{g_c(k-1,n)}$ (tower of exponentiations of height k)
Deciding $K \vDash \varphi$ for alternation depth k is in $\mathbf{NSPACE}(\mathbf{g(k, |\varphi|)})$ and $\mathbf{NSPACE}(\mathbf{g(k-1, |K|)})$ (where NSPACE$(g(-1, n))$ = NLOGSPACE for convenience).
$\Longrightarrow$ Every alternation adds exponential

Exercise

Whats the Alternation-Depth?

- $\varphi_1 = (\forall \pi.\ \mathbf{G}\ a_\pi) \wedge (\exists \pi'.\ \mathbf{F}\ \neg a_{\pi'})$
- $\varphi_2 = \forall \pi_1.\ \exists \pi_2.\ \mathbf{G}\ (a_{\pi_1} \leftrightarrow \mathbf{X}\ b_{\pi_2})$
- $\varphi_3 = \exists \pi.\ \mathbf{X}\ \exists \pi'.\ \mathbf{G}\ (a_\pi \wedge b_{\pi'})$
- $\varphi_4 = \forall \pi_1.\ a_{\pi_1} \wedge \mathbf{X}\ \exists\ \pi_2.(\forall \pi_3.\ a_{\pi_3} \implies (a_{\pi_2} \wedge b_{\pi_3}))$
- $\varphi_5 = \exists \pi.(\mathbf{X}\ \exists \pi'.a_{\pi'})\mathbf{U}\ a_\pi$
- $\forall \pi_1.\ \exists \pi_2.\ (\mathbf{G}\ \lambda_{\pi_2}) \wedge \pi_1 =_L \pi_2$ (Noninference)
- $\forall \pi_1.\ \forall \pi_2.\ \pi_1[0] =_{L,in} \pi_2[0] \implies \pi_1 =_{L,out} \pi_2$ (Observational determinism)

Exercise

Whats the Alternation-Depth?

- $\varphi_1 = (\forall \pi.\ \mathbf{G}\ a_\pi) \wedge (\exists \pi'.\ \mathbf{F}\ \neg a_{\pi'})$ depth: 0
- $\varphi_2 = \forall \pi_1.\ \exists \pi_2.\ \mathbf{G}\ (a_{\pi_1} \leftrightarrow \mathbf{X}\ b_{\pi_2})$
- $\varphi_3 = \exists \pi.\ \mathbf{X}\ \exists \pi'.\ \mathbf{G}\ (a_\pi \wedge b_{\pi'})$
- $\varphi_4 = \forall \pi_1.\ a_{\pi_1} \wedge \mathbf{X}\ \exists\ \pi_2.(\forall \pi_3.\ a_{\pi_3} \implies (a_{\pi_2} \wedge b_{\pi_3}))$
- $\varphi_5 = \exists \pi.(\mathbf{X}\ \exists \pi'.a_{\pi'})\mathbf{U}\ a_\pi$
- $\forall \pi_1.\ \exists \pi_2.\ (\mathbf{G}\ \lambda_{\pi_2}) \wedge \pi_1 =_L \pi_2$ (Noninference)
- $\forall \pi_1.\ \forall \pi_2.\ \pi_1[0] =_{L,in} \pi_2[0] \implies \pi_1 =_{L,out} \pi_2$ (Observational determinism)

Exercise

Whats the Alternation-Depth?

- $\varphi_1 = (\forall \pi.\ \mathbf{G}\ a_\pi) \wedge (\exists \pi'.\ \mathbf{F}\ \neg a_{\pi'})$ — depth: 0
- $\varphi_2 = \forall \pi_1.\ \exists \pi_2.\ \mathbf{G}\ (a_{\pi_1} \leftrightarrow \mathbf{X}\ b_{\pi_2})$ — depth: 1
- $\varphi_3 = \exists \pi.\ \mathbf{X}\ \exists \pi'.\ \mathbf{G}\ (a_\pi \wedge b_{\pi'})$
- $\varphi_4 = \forall \pi_1.\ a_{\pi_1} \wedge \mathbf{X}\ \exists\ \pi_2.(\forall \pi_3.\ a_{\pi_3} \implies (a_{\pi_2} \wedge b_{\pi_3}))$
- $\varphi_5 = \exists \pi.(\mathbf{X}\ \exists \pi'.a_{\pi'})\mathbf{U}\ a_\pi$
- $\forall \pi_1.\ \exists \pi_2.\ (\mathbf{G}\ \lambda_{\pi_2}) \wedge \pi_1 =_L \pi_2$ (Noninference)
- $\forall \pi_1.\ \forall \pi_2.\ \pi_1[0] =_{L,in} \pi_2[0] \implies \pi_1 =_{L,out} \pi_2$ (Observational determinism)

Exercise

Whats the Alternation-Depth?

- $\varphi_1 = (\forall \pi.\ \mathbf{G}\ a_\pi) \wedge (\exists \pi'.\ \mathbf{F}\ \neg a_{\pi'})$ — depth: 0
- $\varphi_2 = \forall \pi_1.\ \exists \pi_2.\ \mathbf{G}\ (a_{\pi_1} \leftrightarrow \mathbf{X}\ b_{\pi_2})$ — depth: 1
- $\varphi_3 = \exists \pi.\ \mathbf{X}\ \exists \pi'.\ \mathbf{G}\ (a_\pi \wedge b_{\pi'})$ — depth: 0
- $\varphi_4 = \forall \pi_1.\ a_{\pi_1} \wedge \mathbf{X}\ \exists\ \pi_2.(\forall \pi_3.\ a_{\pi_3} \implies (a_{\pi_2} \wedge b_{\pi_3}))$
- $\varphi_5 = \exists \pi.(\mathbf{X}\ \exists \pi'.a_{\pi'})\mathbf{U}\ a_\pi$
- $\forall \pi_1.\ \exists \pi_2.\ (\mathbf{G}\ \lambda_{\pi_2}) \wedge \pi_1 =_L \pi_2$ (Noninference)
- $\forall \pi_1.\ \forall \pi_2.\ \pi_1[0] =_{L,in} \pi_2[0] \implies \pi_1 =_{L,out} \pi_2$ (Observational determinism)

Exercise

Whats the Alternation-Depth?

- $\varphi_1 = (\forall \pi.\ \mathbf{G}\ a_\pi) \wedge (\exists \pi'.\ \mathbf{F}\ \neg a_{\pi'})$ — depth: 0
- $\varphi_2 = \forall \pi_1.\ \exists \pi_2.\ \mathbf{G}\ (a_{\pi_1} \leftrightarrow \mathbf{X}\ b_{\pi_2})$ — depth: 1
- $\varphi_3 = \exists \pi.\ \mathbf{X}\ \exists \pi'.\ \mathbf{G}\ (a_\pi \wedge b_{\pi'})$ — depth: 0
- $\varphi_4 = \forall \pi_1.\ a_{\pi_1} \wedge \mathbf{X}\ \exists\ \pi_2.(\forall \pi_3.\ a_{\pi_3} \implies (a_{\pi_2} \wedge b_{\pi_3}))$ — depth: 2
- $\varphi_5 = \exists \pi.(\mathbf{X}\ \exists \pi'.a_{\pi'})\mathbf{U}\ a_\pi$
- $\forall \pi_1.\ \exists \pi_2.\ (\mathbf{G}\ \lambda_{\pi_2}) \wedge \pi_1 =_L \pi_2$ (Noninference)
- $\forall \pi_1.\ \forall \pi_2.\ \pi_1[0] =_{L,in} \pi_2[0] \implies \pi_1 =_{L,out} \pi_2$ (Observational determinism)

Exercise

Whats the Alternation-Depth?

- $\varphi_1 = (\forall \pi.\ \mathbf{G}\ a_\pi) \wedge (\exists \pi'.\ \mathbf{F}\ \neg a_{\pi'})$ — depth: 0
- $\varphi_2 = \forall \pi_1.\ \exists \pi_2.\ \mathbf{G}\ (a_{\pi_1} \leftrightarrow \mathbf{X}\ b_{\pi_2})$ — depth: 1
- $\varphi_3 = \exists \pi.\ \mathbf{X}\ \exists \pi'.\ \mathbf{G}\ (a_\pi \wedge b_{\pi'})$ — depth: 0
- $\varphi_4 = \forall \pi_1.\ a_{\pi_1} \wedge \mathbf{X}\ \exists\ \pi_2.(\forall \pi_3.\ a_{\pi_3} \implies (a_{\pi_2} \wedge b_{\pi_3}))$ — depth: 2
- $\varphi_5 = \exists \pi.(\mathbf{X}\ \exists \pi'.a_{\pi'})\mathbf{U}\ a_\pi$ — depth: 1
- $\forall \pi_1.\ \exists \pi_2.\ (\mathbf{G}\ \lambda_{\pi_2}) \wedge \pi_1 =_L \pi_2$ (Noninference)
- $\forall \pi_1.\ \forall \pi_2.\ \pi_1[0] =_{L,in} \pi_2[0] \implies \pi_1 =_{L,out} \pi_2$ (Observational determinism)

Exercise

Whats the Alternation-Depth?

- $\varphi_1 = (\forall \pi.\ \mathbf{G}\ a_\pi) \wedge (\exists \pi'.\ \mathbf{F}\ \neg a_{\pi'})$ — depth: 0
- $\varphi_2 = \forall \pi_1.\ \exists \pi_2.\ \mathbf{G}\ (a_{\pi_1} \leftrightarrow \mathbf{X}\ b_{\pi_2})$ — depth: 1
- $\varphi_3 = \exists \pi.\ \mathbf{X}\ \exists \pi'.\ \mathbf{G}\ (a_\pi \wedge b_{\pi'})$ — depth: 0
- $\varphi_4 = \forall \pi_1.\ a_{\pi_1} \wedge \mathbf{X}\ \exists\ \pi_2.(\forall \pi_3.\ a_{\pi_3} \implies (a_{\pi_2} \wedge b_{\pi_3}))$ — depth: 2
- $\varphi_5 = \exists \pi.(\mathbf{X}\ \exists \pi'.a_{\pi'})\mathbf{U}\ a_\pi$ — depth: 1
- $\forall \pi_1.\ \exists \pi_2.\ (\mathbf{G}\ \lambda_{\pi_2}) \wedge \pi_1 =_L \pi_2$ (Noninference) — depth: 1
- $\forall \pi_1.\ \forall \pi_2.\ \pi_1[0] =_{L,in} \pi_2[0] \implies \pi_1 =_{L,out} \pi_2$ (Observational determinism)

Exercise

Whats the Alternation-Depth?

- $\varphi_1 = (\forall \pi.\ \mathbf{G}\ a_\pi) \wedge (\exists \pi'.\ \mathbf{F}\ \neg a_{\pi'})$ — depth: 0
- $\varphi_2 = \forall \pi_1.\ \exists \pi_2.\ \mathbf{G}\ (a_{\pi_1} \leftrightarrow \mathbf{X}\ b_{\pi_2})$ — depth: 1
- $\varphi_3 = \exists \pi.\ \mathbf{X}\ \exists \pi'.\ \mathbf{G}\ (a_\pi \wedge b_{\pi'})$ — depth: 0
- $\varphi_4 = \forall \pi_1.\ a_{\pi_1} \wedge \mathbf{X}\ \exists\ \pi_2.(\forall \pi_3.\ a_{\pi_3} \implies (a_{\pi_2} \wedge b_{\pi_3}))$ — depth: 2
- $\varphi_5 = \exists \pi.(\mathbf{X}\ \exists \pi'.a_{\pi'})\mathbf{U}\ a_\pi$ — depth: 1
- $\forall \pi_1.\ \exists \pi_2.\ (\mathbf{G}\ \lambda_{\pi_2}) \wedge \pi_1 =_L \pi_2$ (Noninference) — depth: 1
- $\forall \pi_1.\ \forall \pi_2.\ \pi_1[0] =_{L,in} \pi_2[0] \implies \pi_1 =_{L,out} \pi_2$ (Observational determinism) — depth: 0

Outline

2 ω-Regular Languages and LTL Model Checking

3 HyperLTL and HyperCTL*

4 Automata-Theoretic Approach for checking HyperCTL*

5 Checking Circuits and Industrial Case Studies
- Symbolic Model Checking of Circuits
- Case Studies and Experimental Evaluation

6 Questions?

Symbolic Model Checking of Circuits [1]

Why Symbolic Circuit Model Checking?

- Automata-theoretic model checking for hyperproperties was strictly **explicit-state**
- An explicit circuit with $|X|$ binary states induces an underlying transition system of exponential size ($2^{|X|}$)

Symbolic Model Checking of Circuits [1]

Why Symbolic Circuit Model Checking?

- Automata-theoretic model checking for hyperproperties was strictly **explicit-state**
- An explicit circuit with $|X|$ binary states induces an underlying transition system of exponential size ($2^{|X|}$)

Goal

- Instead of building state graphs or explicit automata, compile the system and the specification directly into a **single combined hardware circuit** C_φ
- The resulting circuit C_φ remains **linear** in both the circuit size and the formula size

Symbolic Model Checking of Circuits

Formal Model of Hardware Circuit

A sequential hardware circuit is a tuple $C = (X, init, I, O, T)$ with

- X: a finite set of binary state variables (latches with unit delay)
- $init \in \mathcal{B}(X)$: A Boolean formula over X characterizing the non-empty set of initial states
- I: A set of primary input variables
- O: A set of primary output variables
- $T \in \mathcal{B}(X \times I \times O \times X')$: A Boolean transition relation relating the current latch values x, inputs i, outputs o and next latch values x'
- **Assumptions:** T is **input-enabled** and **input-deterministic**

Symbolic Model Checking of Circuits

Circuit Transformation: Compiling Logic into Wires

Given circuit C and an alternation-free formula φ (in NNF with $\exists$ quantifiers), we synthesize an augmented circuit C_φ bottom-up. Each subformula ψ introduces three signals:

- o_ψ **(Output wire)**: Asserts that ψ must hold for the current computation path at the current step
- x_ψ **(Memory latch)**: A unit-delay latch storing pending temporal obligations for future computation steps
- b_ψ **(Bad wire / Violation flag)**: Becomes 1 if the requirements for subformula ψ are violated.

Symbolic Model Checking of Circuits

Circuit Transformation: Compiling Logic into Wires

Given circuit C and an alternation-free formula φ (in NNF with $\exists$ quantifiers), we synthesize an augmented circuit C_φ bottom-up. Each subformula ψ introduces three signals:

- o_ψ **(Output wire)**: Asserts that ψ must hold for the current computation path at the current step
- x_ψ **(Memory latch)**: A unit-delay latch storing pending temporal obligations for future computation steps
- b_ψ **(Bad wire / Violation flag)**: Becomes 1 if the requirements for subformula ψ are violated.

Dual Verification for Universal Specifications

To verify a universal property $\varphi = \forall \pi_1 \dots \forall \pi_k . \psi$ (e.g., Noninterference, Symmetry):

- Compile the **negated formula** $\neg\varphi \equiv \exists \pi_1 \dots \exists \pi_k . \neg\psi$ into circuit $C_{\neg\varphi}$.
- **Counterexample (×):** Finding a fair path in $C_{\neg\varphi}$ disproves φ.
- **Proof (✓):** Proving no fair path exists in $C_{\neg\varphi}$ proves $C \vDash \varphi$.

Circuit Transformation: Encoding Inductive Rules

- $\psi = a_{\pi_k}$: Output wire $o_\psi \leftrightarrow a_{\pi_k}$ (no latches).
- $\psi = \psi_1 \vee \psi_2$: Fresh input i_ψ guesses which disjunct holds: $o_\psi \leftrightarrow (i_\psi \Rightarrow o_{\psi_1}) \wedge (\neg i_\psi \Rightarrow o_{\psi_2})$.
- $\psi = \mathbf{X}\ \psi_1$: Output wire $o_\psi \leftrightarrow i_\psi$, latch $x'_\psi \leftrightarrow i_\psi$, bad wire $\neg b_\psi \leftrightarrow (o_{\psi_1} \leftrightarrow x_\psi)$.
- $\psi = \exists \pi.\, \psi_1$: Adds a fresh copy X_n of the circuit latches and transition relation $T(X_n)$, synchronizing initial states $X_n = X_{n-1}$ when o_ψ starts!

Symbolic Model Checking of Circuits

Circuit Transformation: Encoding Inductive Rules

- $\psi = a_{\pi_k}$: Output wire $o_\psi \leftrightarrow a_{\pi_k}$ (no latches).
- $\psi = \psi_1 \vee \psi_2$: Fresh input i_ψ guesses which disjunct holds: $o_\psi \leftrightarrow (i_\psi \Rightarrow o_{\psi_1}) \wedge (\neg i_\psi \Rightarrow o_{\psi_2})$.
- $\psi = \mathbf{X}\ \psi_1$: Output wire $o_\psi \leftrightarrow i_\psi$, latch $x'_\psi \leftrightarrow i_\psi$, bad wire $\neg b_\psi \leftrightarrow (o_{\psi_1} \leftrightarrow x_\psi)$.
- $\psi = \exists \pi.\, \psi_1$: Adds a fresh copy X_n of the circuit latches and transition relation $T(X_n)$, synchronizing initial states $X_n = X_{n-1}$ when o_ψ starts!

Circuit Transformation: Linear Size Guarantee

Circuit size grows **linearly** with both circuit latches and formula size: $Size(C_\varphi) \leq |C| \cdot k + |\varphi|$

Circuit Transformation: Example Simple Sequential Hardware Circuit

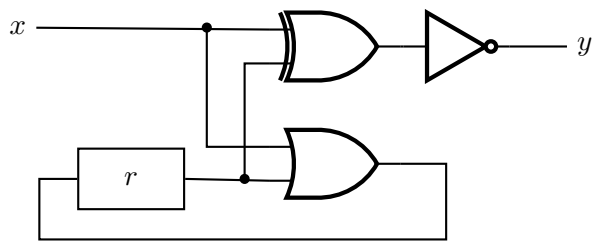


Figure 1 Sequential circuit $C = (X, init, I, O, T)$ with input x, latch r, output $y = \neg(x \oplus r)$, and transition $r' = x \vee r$.

Circuit Transformation: Example Simple Sequential Hardware Circuit

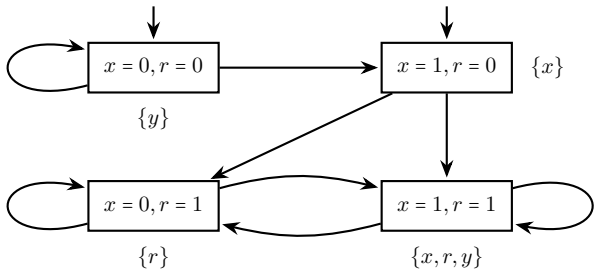


Figure 2 Induced explicit Kripke structure K_C with 4 states (exponential blowup $2^{|X|}$ in general).

Reduction to Fair Reachability

$C \vDash \varphi$ holds if and only if C_φ admits an infinite execution path such that

- **1. Initialization:** Asserts output $o_\varphi = 1$ in the very first clock cycle
- **2. Safety: Never** asserts the bad signal ($b_\varphi = 0$ for all subformulas at all time steps)
- **3. Fairness:** Satisfies all strong fairness constraints $F = \{f_1, \ldots, f_m\}$ where each $f = (a_1, a_2)$ requires a_1 holds finitely often or (a_2) holds infinitely often

$\Longrightarrow$ The entire hyperproperty verification task is now reduced to **standard circuit fair reachability**

Symbolic Model Checking of Circuits

Industrial Solver Integration

The output is emitted as an **AIGER** circuit, feeding directly into the **ABC verification suite**. Modern SAT-based engines deployed:

- **BMC (Bounded Model Checking):** Unrolls transition relations to find counter examples rapidly
- **IC3 / PDR (Property Directed Reachability):** Synthesizes stepwise inductive invariants across latches without unrolling.
- **Interpolation (INT):** Computes Craig interpolants over unsatisfiable SAT unrollings to prove unbounded safety

Case Studies and Experimental Evaluation [1]

Case Study 1: Information Flow Properties of I^2C

- **System:** An open-source implementation of an I^2C **bus master**
- **Complexity:** Synthesized into an AIGER sequential circuit comprising **254 latches** and **1,207 logic gates**
- Interface:
  1. **Master-to-Slave:** Communicates via two open-drain physical lines: serial clock (**SCL**) and serial data (**SDA**)
  2. **Master-to-Controller:** 8-bit bidirectional data lines, 3-bit slave address line, a system clock, write enable and various control/reset signals

Case Study 1: Information Flow Properties of I^2C

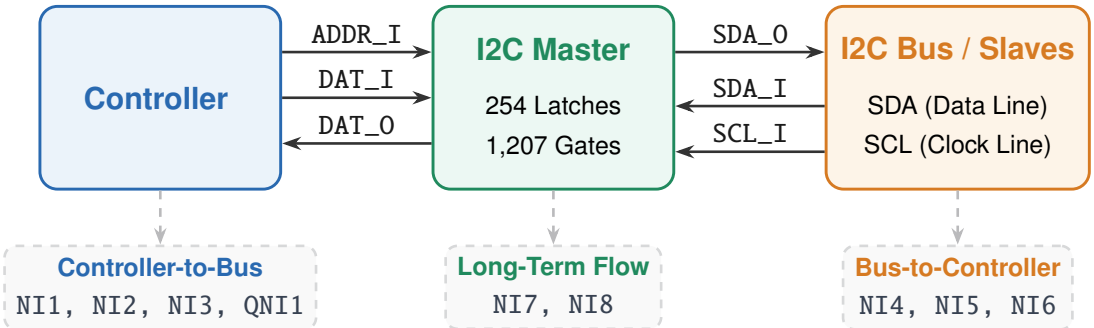


Figure 3 Classification of information flows in the I2C Master communication architecture.

Case Studies and Experimental Evaluation

Case Study 1: Information Flow Properties of I^2C

8 information-flow properties across three categories

1. **Controller-to-Bus:** Target Address Flow (`NI1`), Data Flow (`NI2`), Write-Gated Transmission (`NI3`), Quantitative Information Flow (`QNI1` & `QNI2`)
2. **Bus-to-Controller:** Slave-to-Controller Flow (`NI4` & `NI6`), Concurrent Arbitration and Information Hiding (`NI5`)
3. **Long-term information flow:** Arbitrary Delay Non-Interference (`NI7`, `NI8`)

Case Studies and Experimental Evaluation

Case Study 1: Verification Outcomes

- **Expected Failures** (`NI1`, `NI2`, `NI4`): Correctly violated in **under 0.1 seconds via BMC**
- **Proving Security Invariants** (`NI3`): The write-gated property **satisfied the specification**
- **Uncovering Unintended Side Channels** (`NI5`): **Failed**.
- **Information Persistence** (`NI7`, `NI8`): Both properties were **violated**, revealing that the I^2C master does not eventually clear or forget past inputs.

Case Study 2: Symmetry in Mutual Exclusion Protocols

- **Problem:** Traditional LTL only checks single-trace abstractions (*accessibility, bounded overtaking*), which cannot guarantee **true fairness**
- **Objective:** Verify **symmetry access** - ensuring no client has an unfair structural advantage over others
- **System:** A Verilog implementation of **Bakery Protocol** where contending processes draw tickets numbers; ties are broken by comparing process IDs
- **HyperLTL Formalization:**
  $\forall\pi.\forall\pi'.\mathbf{G}(sym(select_\pi, select_{\pi'}) \wedge (pause_\pi = pause_{\pi'})) \Rightarrow \mathbf{G}(pc(0)_\pi = pc(1)_{\pi'} \wedge pc(1)_\pi = pc(0)_{\pi'})$

Case Studies and Experimental Evaluation

Case Study 2: Iterative Debugging Workflow

1. **Unassigned IDs bug:** Checking $S1$ immediately produced a counterexample in **0.08s via BMC**
2. **Valid IDs only ($S2$, adding $select < 3$):** Still failed since tie-breaking hardcodes priority for process 0
3. **Symmetric Tie-Breaker ($S3$ & RTL fix):** Satisfied after contracting ID comparison into a single step
4. **Property Restructuring:** Rewriting the property from an implication to weak until ($\psi W \neg\varphi$) eliminate liveness constraints, enabling scaling up to **7 concurrent process**

Case Study 3: Error Resistant Codes

- **Motivation:** Error-correcting codes require that all valid codewords maintain a **minimum Hamming distance** d across the codeword space
- **System:** Evaluated on a Huffman encoder, an **8b/10b encoder** and an **8b/10b decoder**
- **HyperLTL Formalization:**

$$\forall \pi . \forall \pi' . \mathbf{F} \left(\bigvee_{a \in I} a_\pi \neq a_{\pi'} \right) \implies \neg \mathsf{Ham}_O(d-1, \pi, \pi')$$

Case Studies and Experimental Evaluation

Case Study 3: Verification Evaluation

- **Sanity Checking Encoders:** Standard Huffman and 8b/10b encoders satisfied $d = 1$ (all distinct inputs produce distinct outputs) but failed for $d > 1$.
- **Detecting Control Flaws:** Verifying the plain property on 8b/10b encoder revealed that the reset signal must be set to false before sending data (otherwise output collisions occur)
- **Decoder Injective Mapping:** Verified the plain property on the 8b/10b decoder circuit, proving that two codewords differing by 1 bit map to distinct decoded output
- **7-4 Hamming Encoder:** BMC found a counterexample in **0.02s** where distinct inputs produced identical zero outputs — uncovering an **active high** vs. **active-low** $reset$ bug in the RTL implementation!

Case Studies and Experimental Evaluation

Benchmark	#Lat	INT (s)	IC3 (s)	BMC (s)	Res
Hamm3 (HD1$_3$)	27	0.04	0.03	**0.02**	×
Hamm7 (HD3)	27	**1.25**	12.23	–	✓

Question: Solver Selection

Why did **BMC** finish in 0.02s in Hamm3 ($d = 1$ with reset bug) but could **not** terminate (–) on Hamm7 ($d = 3$, bug fixed)?

Case Studies and Experimental Evaluation

Benchmark	#Lat	INT (s)	IC3 (s)	BMC (s)	Res
Hamm3 ($\texttt{HD1}_3$)	27	0.04	0.03	**0.02**	×
Hamm7 (HD3)	27	**1.25**	12.23	–	✓

Question: Solver Selection

Why did **BMC** finish in 0.02s in Hamm3 ($d = 1$ with reset bug) but could **not** terminate (–) on Hamm7 ($d = 3$, bug fixed)?

Solution

- **BMC** only searches for **bounded counter examples** (bugs) - it quickly finds the short reset-bug in Hamm3, but cannot prove that no counterexample exists at *any* depth

Case Studies and Experimental Evaluation

Benchmark	#Lat	INT (s)	IC3 (s)	BMC (s)	Res
Hamm3 ($\texttt{HD1}_3$)	27	0.04	0.03	**0.02**	×
Hamm7 (HD3)	27	**1.25**	12.23	–	✓

Question: Solver Selection

Why did **BMC** finish in 0.02s in Hamm3 ($d = 1$ with reset bug) but could **not** terminate (–) on Hamm7 ($d = 3$, bug fixed)?

Solution

- **BMC** only searches for **bounded counter examples** (bugs) - it quickly finds the short reset-bug in Hamm3, but cannot prove that no counterexample exists at *any* depth
- Once fixed (Hamm7), proving the property holds over **infinite runs** requires synthesizing inductive invariant via **IC3**/**PDR** or **Interpolation (INT)**.

Case Studies and Experimental Evaluation

Benchmark	#Lat	INT (s)	IC3 (s)	BMC (s)	Res
I2C (NI1)	254	1.13	95.17	**0.07**	×
I2C (NI3)	254	**1.38**	168.96	–	✓
I2C (NI5)	254	8.31	717.74	**0.77**	×
Ethernet (NI2')	**21,093**	155.77	TO	**6.27**	×
Bakery (S1)	46	0.88	6.34	**0.08**	×
Bakery.a.n.s (S3)	47	TO	**66.41**	–	✓
Bakery.5proc (S6)	90	TO	**13.59**	–	✓
Bakery.7proc (S7)	136	TO	**312.53**	–	✓
8b10b_enc (HD1')	39	9.06	**1.19**	–	✓
Hamm3 ($\mathtt{HD1}_3$)	27	0.04	0.03	**0.02**	×
Hamm3' ($\mathtt{HD1}'_3$)	27	**0.18**	7.34	–	✓
Hamm7 (HD3)	27	**1.25**	12.23	–	✓

Key Takeaway from the Table

- **BMC** is fastest at finding counterexamples (×), scaling to circuits with **over 20,000 latches** (e.g., Ethernet core in 6.27 s).
- **IC3** / **Interpolation (INT)** successfully prove invariants (✓) on designs with **hundreds of latches** (e.g., I2C and Bakery protocols).

Any Questions about the topics?

Sources

[1] Bernd Finkbeiner Markus N. Rabe César Sánchez. “Algorithms for Model Checking HyperLTL and HyperCTL*”. In: (2015).

[2] Joost-Pieter Katoen Christel Baier. *Principles of Model Checking*. The MIT Press, 2008.

[3] Takeshi Hayashi Satoru Miyano. “Alternating Finite Automata on ω-Words”. In: (1983).