Algorithms for Model Checking HyperLTL and HyperCTL*

Bernd Finkbeiner[1], Markus N. Rabe[1], and César Sánchez[2]

[1]Saarland University, [2]IMDEA Software Institute

Abstract. We present an automata-based algorithm for checking finite state systems for hyperproperties specified in HyperLTL and HyperCTL*. For the alternation-free fragments of HyperLTL and HyperCTL* the automaton construction allows us to leverage existing model checking technology. Along several case studies, we demonstrate that the approach enables the verification of real hardware designs for properties that could not be checked before. We study information flow properties of an I2C bus master, the symmetric access to a shared resource in a mutual exclusion protocol, and the functional correctness of encoders and decoders for error resistant codes.

1 Introduction

HyperLTL and HyperCTL* are recent extensions to LTL and CTL* with the ability to express a wide range of hyperproperties [14]. Hyperproperties generalize trace properties and include properties from information-flow security such as noninterference [15]. Even though the complexity of model checking HyperLTL and HyperCTL* has been determined, no efficient algorithms are known so far. In this paper, we thus study the automatic verification of finite state systems for hyperproperties specified in HyperLTL and HyperCTL*.

HyperLTL and HyperCTL* allow us to specify relations over executions of the same system [14]. They introduce path quantifiers so computation paths can be referred to in the atomic propositions. For example, the following HyperLTL formula expresses noninterference [22] between input h and output o by requiring that all computation paths π and π' that only differ in h, have the same output o at all times:

$$\forall\pi.\forall\pi'.\ \square\big(\bigwedge_{i\in I\setminus h} i_\pi = i_{\pi'}\big)\ \Rightarrow\ \square(o_\pi = o_{\pi'})$$

Quantifiers in CTL*, in contrast, are of the form $\mathsf{A}\varphi$ and $\mathsf{E}\varphi$ where the subformula φ can only (implicitly) refer to a single path—the path introduced by A and E respectively. Hence, CTL* cannot express noninterference [20,1].

This work was partially supported by the Spanish Ministry of Economy under project "TIN2012-39391-C04-01 STRONGSOFT," the Madrid Regional Government under the project "S2013/ICE-2731 N-Greens Software-CM," the German Research Foundation (DFG) under the project SpAGAT in the Priority Program 1496 "Reliably Secure Software Systems - RS3," and the Graduate School of Computer Science at Saarland University.

Noninterference between i and o implies that o contains no information about i, and is therefore an important building block for properties in security [23]. By embedding noninterference in a temporal context, HyperLTL and HyperCTL* allow us to express a wide range of properties from information-flow security, including variants of declassification and quantitative information flow [3,5,16,42]. The use cases of HyperLTL and HyperCTL*, however, extend far beyond security, as we demonstrate in this paper.

The main result of this paper is an automata-theoretic algorithm for the model checking problem of HyperLTL and HyperCTL*. The automata approach to model checking LTL properties [47] reduces the verification problem to automata operations and decision problems, like automata product and check for emptiness. Typically, the LTL specification is translated into a Büchi word automaton that captures all violations of the specification. The product of the system with this automaton reveals the system's traces that violate the specification. We extend the approach based on Büchi word automata with the ability to quantify over new executions along the run, and thereby obtain an algorithm for HyperCTL* (Section 3). The construction for a quantifier $\exists \pi.\ \varphi$ corresponds to a product of the system and the automaton for the subformula φ. As in the classical approach, a final check of emptiness of the language of the automaton provides the answer to the model checking problem. The construction of the automaton involves the expensive nondeterminization of alternating automata [37] to handle quantifier alternations. For the rich class of alternation-free formulas, however, the algorithm is shown to be in NLOGSPACE in the size of the system. In Section 4 we use the alternating automaton construction to derive an approach to leverage existing model checking technology for model checking circuits for the alternation-free fragment of HyperCTL*.

We demonstrate the flexibility and the effectiveness of the proposed approach for the alternation-free fragment of HyperCTL* along three case studies (Section 5). The first case study concerns the information flow analysis of an I2C bus master. The second case study concerns the analysis of the symmetries in a mutual exclusion protocol. The typical fair-access properties against which mutual exclusion protocols are usually analyzed, such as accessibility and bounded overtaking [31], can be seen as abstractions of what is really expected from mutual exclusion protocols: *symmetric* access to the shared resource. HyperLTL enables a fine grained analysis of the symmetry between the processes, for example by expressing the property that switching the actions and roles between two components in a trace results in another legal trace, in which the access to the shared resource is switched accordingly. The third case study concerns the functional correctness of encoders and decoders of error resistant codes. The error resistance of a code is a property of its space of code words: all pairs of code words must have a certain minimum Hamming distance. We show that Hamming distance can be expressed in HyperLTL and demonstrate that this leads to an effective approach to the verification of encoders and decoders.

To summarize, our contributions are as follows:

- We develop the first direct automaton construction for model checking HyperLTL and HyperCTL* based on alternating automata.

– We present the first practical approach for model checking hardware systems for alternation-free HyperCTL* formulas.

Our evaluation shows that the approach enables the verification of industrial size hardware modules for hyperproperties. That is, we extend the state of the art in model checking hyperproperties from systems using only few (binary) variables [14,35] to systems with over 20.000 variables.

Related work. In this paper, we present an automata-theoretic model checking algorithm for HyperLTL and HyperCTL*, together with a practical approach to the verification of hardware circuits against alternation-free formulas. Previous automata constructions for the problem [14] are based on *nondeterministic* Büchi automata, whereas we present an algorithm based on *alternating* Büchi automata, which allows us to leverage modern hardware verification techniques like IC3 [10]/PDR [18], interpolation [33], and SAT [8]. Our model checker can therefore be applied to significantly more complex systems than the proof-of-concept model checker for the one-alternation fragment of HyperLTL [14], which is limited to small explicitly given models.

HyperLTL and HyperCTL* are related to other logics for hyperproperties, such as variations of the μ-calculus, like the polyadic μ-calculus by Andersen [2], the higher-dimensional μ-calculus [39], and holistic hyperproperties [36]. The model checking problem for these logics can be reduced to the model checking problem of the modal μ-calculus [2,28] (or directly to parity games [35]) and involves, similar to our construction, an analysis of the product of several copies of the system. We are not aware, however, of any practical approaches that would allow the verification of complex hardware designs against specifications given in these logics. Another related class of logics are the epistemic temporal logics [19], which reason about the *knowledge of agents* and how it changes over time. While it has been shown that epistemic temporal logic can express certain information flow policies [4], most practical work with epistemic logics has focussed on applications from the area of multi-agent systems [21,29,30,34,40].

Lastly, in the area of information flow security, there are several verification techniques that focus on specific information flow properties—rather than on a general logic like HyperLTL and HyperCTL*—but use techniques that relate to our model checking algorithm. A construction based on the product of copies of a system, self-composition [6,7], has been tailored for various trace-based security definitions [17,24,45].

2 Temporal Logics for Hyperproperties

We now introduce the temporal logics for hyperproperties, their semantics, and their model checking problem.

A *Kripke structure* is a tuple $K = (S, s_0, \delta, \mathsf{AP}, L)$ consisting of a set of states S, an initial state s_0, a transition function $\delta : S \to 2^S$, a set of *atomic propositions* AP, and a *labeling function* $L : S \to 2^{\mathsf{AP}}$ decorating each state with a set of atomic propositions. We require that each state has a successor, that is $\delta(s) \neq \emptyset$,

to ensure that every execution of a Kripke structure can always be extended to an infinite execution. A *path* of a Kripke structure is an infinite sequence of states $s_0 s_1 \ldots \in S^\omega$ such that s_0 is the initial state of K and $s_{i+1} \in \delta(s_i)$ for all $i \in \mathbb{N}$. We denote by $\mathit{Paths}(K, s)$ the set of all paths of K starting in state $s \in S$ and by $\mathit{Paths}^*(K, s)$ the set of their suffixes. Given a path p and a number $i \geq 0$, $p[i, \infty]$ denotes the suffix path where the first i elements are removed.

HyperLTL and HyperCTL* extend the standard temporal logics LTL and CTL* by *quantification over path variables*. Their formulas are generated by the following grammar, where $a \in \mathsf{AP}$ and π ranges over path variables:

$$\begin{aligned} \varphi \;::=\; & \mathsf{true} \mid a_\pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \\ & \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \varphi \,\mathcal{R}\, \varphi \mid \exists \pi.\ \varphi \mid \forall \pi.\ \varphi \end{aligned}$$

Additionally, we define the derived operators $\diamondsuit\varphi = \mathsf{true}\ \mathcal{U}\, \varphi$, $\square\varphi = \neg\diamondsuit\neg\varphi$, and $\varphi_1 \,\mathcal{W}\, \varphi_2 = \varphi_1 \,\mathcal{U}\, \varphi_2 \vee \square\varphi_1$.

For HyperLTL and HyperCTL* we require that temporal operators only occur inside the scope of path quantifiers. HyperLTL is the sublogic of formulas in *prenex normal form*. A formula is in prenex normal form, if it starts with a sequence of quantifiers, and is quantifier-free in the rest of the formula. The conceptual difference between HyperLTL and HyperCTL*, is that HyperLTL, like LTL, is a linear-time logic and that HyperCTL*, like CTL and CTL*, is a branching-time logic [20]. A formula φ is in *negation normal form* if the only occurrences of $\neg$ occur in front of propositions a_π.

Semantics. In the following we define the semantics for the operators a_π, $\neg\varphi$, $\varphi_1 \vee \varphi_2$, $\bigcirc\varphi$, $\varphi_1 \,\mathcal{U}\, \varphi_2$, and $\exists\pi.\ \varphi$. The other operators are defined via the following equalities: $\forall\pi.\ \varphi = \neg\exists\pi.\ \neg\varphi$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, and $\varphi_1 \,\mathcal{R}\, \varphi_2 = \neg(\neg\varphi_1 \,\mathcal{U}\, \neg\varphi_1)$. These derived operators are kept in the syntax to guarantee the existence of equivalent formulas in negation normal form.

Let K be a Kripke structure and let s_0 be its initial state. The semantics of HyperLTL and HyperCTL* 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*. We use $\Pi[i, \infty]$ for the map that assigns to each path variable π the suffix $\Pi(\pi)[i, \infty]$. We use the reserved path variable ε to denote the most recently quantified path and define the validity of a formula as follows:

$$\begin{array}{lll}
\Pi \models_K a_\pi & \text{whenever} & a \in L\big(\Pi(\pi)(0)\big) \\
\Pi \models_K \neg\varphi & \text{whenever} & \Pi \not\models_K \varphi \\
\Pi \models_K \varphi_1 \vee \varphi_2 & \text{whenever} & \Pi \models_K \varphi_1 \text{ or } \Pi \models \varphi_2 \\
\Pi \models_K \bigcirc\varphi & \text{whenever} & \Pi[1, \infty] \models_K \varphi \\
\Pi \models_K \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{whenever} & \text{for some } i \geq 0 :\ \Pi[i, \infty] \models_K \varphi_2 \text{ and} \\
 & & \qquad \text{for all } 0 \leq j < i :\ \Pi[j, \infty] \models_K \varphi_1 \\
\Pi \models_K \exists\pi.\ \varphi & \text{whenever} & \text{for some } p \in \mathit{Paths}(K, \Pi(\varepsilon)(0)) : \\
 & & \qquad \Pi[\pi \mapsto p,\ \varepsilon \mapsto p] \models_K \varphi
\end{array}$$

For the empty assignment $\Pi = \{\}$, we define $\Pi(\varepsilon)(0)$ to yield the initial state. Validity on states of a Kripke structure K, written $s \models_K \varphi$, is defined as $\{\} \models_K \varphi$. A Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ satisfies formula φ, denoted with $K \models \varphi$ whenever $s_0 \models_K \varphi$.

3 Automata-Theoretic Model Checking of HyperCTL*

In this section, we present an automata-theoretic construction for the verification of HyperCTL* formulas. In Section 4 we will then use this construction to build a practical algorithm for the verification of circuits. We start with a brief review of alternating automata. Given a finite set Q, $\mathbb{B}(Q)$ denotes the set of Boolean formulas over Q and $\mathbb{B}^+(Q)$ the set of positive Boolean formulas, that is, formulas that do not contain negation. The satisfaction of a formula $\theta \in \mathbb{B}(Q)$ by a set $Q' \subseteq Q$ is denoted by $Q' \models \theta$.

Definition 1 (Alternating Büchi automata). *An* alternating Büchi automaton *(on words) is a tuple $\mathcal{A} = (Q, q_0, \Sigma, \rho, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite alphabet, $\rho : Q \times \Sigma \to \mathbb{B}^+(Q)$ is a transition function that maps a state and a letter to a positive Boolean combination of states, and $F \subseteq Q$ are the accepting states.*

A run of an alternating automaton is a Q-labeled tree. A *tree* T is a subset of $\mathbb{N}^*_{>0}$ such that for every *node* $\tau \in \mathbb{N}^*_{>0}$ and every positive integer $n \in \mathbb{N}_{>0}$, (i) if $\tau \cdot n \in T$ then $\tau \in T$ (i.e., T is prefix-closed), and (ii) for every $0 < m < n$, $\tau \cdot m \in T$. The root of T is the empty sequence ε and for a node $\tau \in T$, $|\tau|$ is the length of the sequence τ, in other words, its distance from the root. A *run* of $\mathcal{A}$ on an infinite word $\pi \in \Sigma^\omega$ is a Q-labeled tree (T, r) such that $r(\varepsilon) = q_0$ and for every node τ in T with children $\tau_1, \ldots, \tau_k$ the following holds: $1 \leq k \leq |Q|$ and $\{r(\tau_1), \ldots, r(\tau_k)\} \models \rho(q, \pi[i])$, where $q = r(\tau)$ and $i = |\tau|$. A run r of $\mathcal{A}$ on $\pi \in \Sigma^\omega$ is *accepting* whenever for every infinite path $\tau_0\tau_1 \ldots$ in T, there are infinitely many i with $r(\tau_i) \in F$. We say that π is accepted by $\mathcal{A}$ whenever there is an accepting run of $\mathcal{A}$ on π, and denote with $\mathcal{L}_\omega(\mathcal{A})$ the set of infinite words *accepted* by $\mathcal{A}$.

If the transition function of an alternating automaton does not contain any conjunctions, we call the automaton *nondeterministic*. The transition function ρ of a nondeterministic automaton thus identifies a disjunction over a set of successor states. Such a transition function can also be stated as a function $\rho : Q \times \Sigma \to 2^Q$ identifying the successors. Our model checking algorithm relies on the standard translation for alternation removal due to Miyano and Hayashi:

Theorem 1 ([37]). *Let $\mathcal{A}$ be an alternating Büchi automaton with n states. There is a nondeterministic Büchi automaton $\mathsf{MH}(\mathcal{A})$ with $2^{\mathcal{O}(n)}$ states that accepts the same language.*

3.1 The Alternation-Free Fragment

We present a model checking algorithm for the alternation-free fragment of HyperCTL*. This fragment is expressive enough to capture a broad range of other information-flow properties, like declassification mechanisms, quantitative noninterference, and information-flow requirements that change over time [14,16]. The case studies in Section 5 illustrate that this fragment also captures properties in application domains beyond information-flow security.

Definition 2 (Alternation-free HyperCTL*). *A HyperCTL* formula φ in negation normal form is* alternation-free*, if φ contains only quantifiers of one type. Additionally, we require that no existential quantifier occurs in the left subformula of an until operator or in the right subformula of a release operator, and, symmetrically, that no universal quantifier occurs in the right subformula of an until operator or in the left subformula of a release operator.*

Similar to the automata-theoretic approach to LTL properties [38,46], we construct an alternating automaton bottom up from the formula, but handling multiple path quantifiers. For alternation-free HyperCTL*, the quantifiers may occur inside temporal operators (with the restrictions in Def. 2) as long as there is no quantifier alternation.

Let K be a Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$. To check the satisfaction of a HyperCTL* formula φ by K, we translate φ into a K-equivalent alternating automaton $\mathcal{A}_\varphi$. The construction of $\mathcal{A}_\varphi$ proceeds inductively following the structure of φ, as follows. Assume that φ is in negation normal form and starts with an existential quantifier, and consider a subformula ψ of φ. Let n be the number of path quantifiers occurring on the path from the root of the syntax tree of φ to ψ, and let these path quantifiers bind the variables $\pi_1, \ldots, \pi_n$. The alphabet Σ of $\mathcal{A}_\psi$ is S^n, the set of n-tuples of states of K. We say that a language $L \subseteq (S^n)^\omega$ is K-*equivalent* to ψ, if all sequences of state tuples $(s_0^0, \ldots, s_n^0)(s_0^1, \ldots, s_n^1) \ldots$ in L correspond to a path assignment Π satisfying ψ. That is, for all $(s_0^0, \ldots, s_n^0)(s_0^1, \ldots, s_n^1) \ldots \in L$ it holds $\Pi \models_K \psi$ for the path assignment $\Pi(\pi_i) = s_i^0 s_i^1 \ldots$ (for all $i \leq n$). An automaton is K-equivalent to ψ if its language is K-equivalent to ψ.

For atomic propositions, Boolean connectives, and temporal operators, our construction follows the standard translation from LTL to alternating automata [38,46]. Let $\mathcal{A}_{\psi_1} = (Q_1, q_{0,1}, \Sigma_1, \rho_1, F_1)$ and $\mathcal{A}_{\psi_2} = (Q_2, q_{0,2}, \Sigma_2, \rho_2, F_2)$ be the alternating automata for the subformulas ψ_1 and ψ_2:

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

For a quantified subformula $\psi = \exists\pi.\psi_1$, we have to reduce the alphabet $\Sigma_{\psi_1} = S^{n+1}$ to $\Sigma = S^n$. The language for formula ψ contains exactly those sequences σ of state tuples, such that there is a path p through the Kripke structure K for which σ extended by p is in $\mathcal{L}(\mathcal{A}_{\psi_1})$. Let $\mathcal{N}'_{\psi_1} = (Q', q'_0, \Sigma, \rho', F')$ be the nondeterministic automaton $\mathcal{N}'_{\psi_1} = \mathsf{MH}(\mathcal{A}_{\psi_1})$ constructed from $\mathcal{A}_{\psi_1}$ by the construction in Theorem 1, and let $\mathcal{A}_\psi = (Q'', q''_0, \Sigma_\psi, \rho'', F'')$ be constructed from $\mathcal{N}'_{\psi_1}$ and the Kripke structure $K = (S, s_0, \delta, \mathsf{AP}, L)$ as follows:

$\psi = \exists\pi.\psi_1$	$\mathcal{A}_\psi = (Q' \times S \cup \{q''_0\}, q''_0, \Sigma_\psi, \rho'', F' \times S)$ where $\rho''(q''_0, \boldsymbol{s}) = \{(q', s') \mid q' \in \rho'(q'_0, \boldsymbol{s} + \boldsymbol{s}|_n), s' \in \delta(\boldsymbol{s}|_n)\}$ and $\rho''((q, s), \boldsymbol{s}) = \{(q', s') \mid q' \in \rho'(q, \boldsymbol{s} + s), s' \in \delta(s)\}$

For the case that $n = 0$ we define that $\boldsymbol{s}|_n$ is the initial state s_0 of K.

Since we consider the alternation-free fragment, there are no negated quantified subformulas and the construction is finished.

The correctness of the construction can be shown by structural induction.

Proposition 1. *Let φ be a HyperCTL* formula and $\mathcal{A}_\varphi$ the alternating automaton obtained by the previous construction. Then, φ and $\mathcal{A}_\varphi$ are K-equivalent.*

So far, we only considered alternation-free formulas that start with existential quantifiers. To decide $K \models \varphi$ for an arbitrary φ, we first transform φ in a Boolean combination over a set X of quantified subformulas. Each element ψ' of X is now in the form $\exists\pi.\varphi$ for which we apply the construction above. Since ψ' is of the form $\exists\pi.\psi_1$, $\mathcal{A}_{\psi'}$ is a nondeterministic Büchi automaton, for which we apply a standard nonemptiness test [48].

Theorem 2. *The model checking problem for the alternation-free fragment of HyperCTL* is PSPACE-complete in the size of the formula and NLOGSPACE-complete in the size of the Kripke structure.*

Proof. The alternating automaton $\mathcal{A}_{\psi_1}$ is a tree with self-loops, when we consider automata created for quantified subformulas as leafs of the tree. By structural induction, we show that the size of $\mathcal{A}_{\psi'}$ for an alternation-free formula ψ' is polynomial in $|\psi'|$ and in $|K|$ and that sub-automata for quantified subformulas are not reachable via actions that are self-loops with conjunctions.
Base case: for atomic propositions and negated atomic propositions, the induction hypothesis is fulfilled.
Induction step: Let $\psi = \exists\pi.\ \psi_1$. Only Until operators and Release operators in the formula lead to nodes that have two transitions, one with a self-loop and one without self-loops. By the restrictions in the definition of the alternation-free fragment, we guarantee that automata of quantified subformulas are not reachable via transitions with self-loops that contain conjunctions.

Conjunctive transitions that are not part of loops or self-loops only lead to a polynomial increase in size during nondeterminization. Emptiness of nondeterministic Büchi automata is in NLOGSPACE [48], so the upper bound of the theorem follows. Since HyperCTL* subsumes LTL, the lower bound for LTL model checking [43] implies the lower bound for HyperCTL*. □

3.2 The Full Logic

The construction from the previous subsection can be extended to full HyperCTL* by adding a construction for *negated* quantified subformulas. We compute an automaton for the complement language, based on the following theorem:

Theorem 3 ([26]). *For every alternating Büchi automaton $\mathcal{A} = (Q, q_0, \Sigma, \rho, F)$, there is an alternating Büchi automaton $\overline{\mathcal{A}}$ with $O(|Q|)^2$ states that accepts the complemented language: $\mathcal{L}_\omega(\overline{\mathcal{A}}) = \overline{\mathcal{L}_\omega(\mathcal{A})}$.*

We extend the previous construction with the following case:

$\varphi = \neg\exists\pi.\psi_1$	$\overline{\mathcal{N}'_{\psi_1}}$, where $\mathcal{N}'_{\psi_1} = \mathsf{MH}(\mathcal{A}_{\psi_1})$ via Theorem 1

We capture the complexity of the resulting model checking algorithm in terms of the alternation depth of the HyperCTL* formula. The formulas with alternation depth 0 are exactly the alternation-free formulas.

Definition 3 (Alternation Depth). *A HyperCTL* formula φ in negation normal form has alternation depth 0 plus the highest number of alternations from existential to universal and universal to existential quantifiers along any of the paths of the formula's syntax tree. Existential quantifiers in the left subformula of an until operator or in the right subformula of a release operator, and, symmetrically, universal quantifiers in the right subformula of an until operator or in the left subformula of a release operator count as additional alternation.*

For example, let ψ be a formula without additional quantifiers, then $\exists\pi.\ \psi$ has alternation depth 0, $\forall\pi_1.\exists\pi.\ \psi$ has alternation depth 1, $\exists\pi.\ \Diamond\exists\pi'.\ \psi$ has alternation depth 0, $\exists\pi.\ \Box\exists\pi'.\ \psi$ has alternation depth 1, and $(\forall\pi.\ \psi) \wedge (\exists\pi.\ \psi)$ has alternation depth 0.

Let $g_c(k, n)$ be a tower of exponentiations of height k, defined simply as $g_c(0, n) = n$ and $g_c(k, n) = c^{g_c(k-1,n)}$. We define NSPACE($g(k, n)$) to be the languages that are accepted by a nondeterministic Turing machine that runs in SPACE $O(g_c(k, n))$ for some $c > 1$. For convenience, we define NSPACE($g(-1, n)$) to be NLOGSPACE.

Proposition 2. *Let K be a Kripke structure and φ a HyperCTL* formula with alternation depth k. The alternating automaton $\mathcal{A}_\varphi$ resulting from the previous construction has $O(g(k+1, |\varphi|))$ and $O(g(k, |K|))$ states and can be constructed in NSPACE($g(k, |\varphi|)$) and NSPACE($g(k-1, |K|)$).*

Theorem 4. *Given a Kripke structure K and a HyperCTL* formula φ with alternation depth k, we can decide whether $K \models \varphi$ in NSPACE($g(k, |\varphi|)$) and NSPACE($g(k-1, |K|)$).*

The proof of Proposition 2 is an induction over the alternation depth. The proof of Theorem 4 uses that the nonemptiness problem for nondeterministic Büchi automata is in NLOGSPACE [48]. Theorem 4 subsumes the result for the alternation-free fragment:

Corollary 1. *For alternation depth 0, the model-checking problem $K \models \varphi$ is in PSPACE in $|\varphi|$ and in NLOGSPACE in $|K|$.*

4 Symbolic Model Checking of Circuits

In this section we translate the automaton-based construction from Section 3 for alternation-free formulas into a practical verification approach for circuits. Given a circuit C and an alternation-free formula φ the algorithm produces a new circuit C_φ that is linear in the size of C and also linear in the size of φ. The compactness of the encoding builds on the ability of circuits to describe systems of exponential size with a linear number of binary variables. The circuit C_φ is then checked for fair reachability to determine the validity of $C \models \varphi$. This check can be done with of-the-shelf model checkers leveraging modern hardware verification technology [12,8,11].

A *circuit* $C = (X, \mathit{init}, I, O, T)$ consists of a set X of binary variables (latches with unit delay), a condition $\mathit{init} \in \mathbb{B}(X)$ characterizing a non-empty set of initial states of X, a set of input variables I, a set of output variables O, and a transition relation $T \in \mathbb{B}(X \times I \times O \times X)$. We require that T is input-enabled and input-deterministic, that is, for all $x \subseteq X$, $i \subseteq I$, there is exactly one $o \subseteq O$ and one $x' \subseteq X$ such that $T(x, i, o, x')$ holds. We denote a subset of X as a *state* of circuit C, indicating exactly those latches that are set to 1. The size of a circuit C, denoted $|C|$, is defined as the number of latches $|X|$.

A circuit C can be interpreted as a finite Kripke structure K_C of potentially exponential size. The state space of K_C is $S = s_0 \cup 2^X \times 2^I \times 2^O \times 2^X$, where s_0 is a fresh initial state. The transition relation distinguishes the initial step of the computation: $s' \in \delta(s_0)$ iff there is a circuit state $x \subseteq X$ with $\mathit{init}(x)$ and $x = s'|_X$ such that $T(x, s'|_I, s'|_O, s'|_X)$, where $s'|_I$, $s'|_O$, $s'|_X$, and $s'|_{X'}$ are the projections to variables I, O, the first copy of X, and the second copy of X respectively. For subsequent steps of computation we define $s' \in \delta(s)$ whenever $T(s|_X, s'|_I, s'|_O, s'|_{X'})$ and $s|_{X'} = s'|_X$. That is, the first copy X denotes the *previous* state, whereas X' denotes the *current* state. The labelling function of K_C maps each state s to the set $s|_I \cup s|_O \cup s|_X$. That is, the alphabet AP_{K_C} is $I \cup O \cup X$. The semantics of HyperCTL* on a circuit C is defined using the associated Kripke structure K_C. We write $C \models \varphi$ whenever $K_C \models \varphi'$, where φ' is obtained by replacing all atomic propositions a_π by $\bigcirc a_\pi$. This leads to a natural semantics on circuits: the atomic propositions always refer to the *current* value of the latches, the *next* input, and the *next* output.

Given a circuit C and an alternation-free HyperCTL* formula φ, we reduce the model checking problem $C \models \varphi$ to finding a computation path in a circuit C_φ that does not visit a bad state and satisfies a conjunction of strong fairness (or compassion) constraints $F = \{f_1, \ldots, f_k\}$. A strong fairness constraint f of a circuit consists of a tuple (a_1, a_2) of atomic propositions and a path p satisfies f, if a_1 holds only finitely often or a_2 holds infinitely often on p. We build C_φ bottom up following the formula structure. Without loss of generality, we assume that φ contains only existential quantifiers and is in negation normal form. Let ψ be a subformula of φ that occurs under n quantifiers. Let $C_{\psi_1} =$

Our definition of circuits can be considered as a model of and-inverter graphs in the Aiger standard [9], where the gate list is abstracted to a transition relation.

$(X_{\psi_1}, \mathit{init}_{\psi_1}, I_{\psi_1}, O_{\psi_1}, T_{\psi_1})$, $C_{\psi_2} = (X_{\psi_2}, \mathit{init}_{\psi_2}, I_{\psi_2}, O_{\psi_2}, T_{\psi_2})$ be the circuits, and let F_{ψ_1} and F_{ψ_2} be the fairness constraints for the subformulas ψ_1 and ψ_2. For LTL operators, the construction resembles the standard translation from LTL to circuits [25,13]. We construct C_ψ and F_ψ as follows:

$\psi = a_{\pi_k}$	$C_\psi = (\emptyset, \mathsf{true}, I_\psi, \{o_\psi\}, o_\psi \leftrightarrow a_{\pi_k})$,	$F_\psi = \emptyset$
$\psi = \neg a_{\pi_k}$	$C_\psi = (\emptyset, \mathsf{true}, I_\psi, \{o_\psi\}, o_\psi \ \mathsf{xor}\ a_{\pi_k})$,	$F_\psi = \emptyset$
$\psi = \psi_1 \vee \psi_2$	$C_\psi = (X_{\psi_1} \dot\cup X_{\psi_2},\ \mathit{init}_{\psi_1} \wedge \mathit{init}_{\psi_2},$ $I_{\psi_1} \cup I_{\psi_2} \dot\cup \{i_\psi\},\ O_{\psi_1} \dot\cup O_{\psi_2} \dot\cup \{o_\psi\},$ $(o_\psi \leftrightarrow (i_\psi \Rightarrow o_{\psi_1}) \wedge (\neg i_\psi \Rightarrow o_{\psi_2})) \wedge T_{\psi_1} \wedge T_{\psi_2})$, $F_\psi = F_{\psi_1} \cup F_{\psi_2}$	
$\psi = \bigcirc \psi_1$	$C_\psi = (X_{\psi_1} \dot\cup \{x_\psi\},\ \mathit{init}_{\psi_1}, I_{\psi_1} \dot\cup \{i_\psi\},\ O_{\psi_1} \dot\cup \{o_\psi, b_\psi\},$ $T_{\psi_1} \wedge (o_\psi \leftrightarrow i_\psi) \wedge (x'_\psi \leftrightarrow i_\psi) \wedge (\neg b_\psi \leftrightarrow (o_{\psi_1} \leftrightarrow x_\psi)))$, $F_\psi = F_{\psi_1}$	
$\psi = \psi_1 \,\mathcal{U}\, \psi_2$	$C_\psi = (X_{\psi_1} \dot\cup X_{\psi_2} \dot\cup \{x_\psi\},\ \mathit{init}_{\psi_1} \wedge \mathit{init}_{\psi_2},$ $I_{\psi_1} \dot\cup I_{\psi_2} \dot\cup \{i_\psi, i'_\psi\},\ O_{\psi_1} \dot\cup O_{\psi_2} \dot\cup \{o_\psi, b_\psi\},$ $T_{\psi_1} \wedge T_{\psi_2} \wedge (o_\psi \leftrightarrow x_\psi) \wedge (x'_\psi \leftrightarrow i_\psi) \wedge$ $(\neg b_\psi \leftrightarrow (((i'_\psi \Rightarrow o_{\psi_2}) \wedge (\neg i'_\psi \Rightarrow o_{\psi_1} \wedge x'_\psi)) \leftrightarrow x_\psi)))$, $F_\psi = F_{\psi_1} \cup F_{\psi_2} \cup \{(x_\psi, o_{\psi_2})\}$	
$\psi = \exists \pi.\ \psi_1$	$C_\psi = (X_{\psi_1} \dot\cup X_n,\ \mathit{init}_{\psi_1} \wedge (n = 1 \Rightarrow \mathit{init}(X_n)),$ $I_{\psi_1} \setminus X_n,\ (O_{\psi_1} \setminus O_n) \dot\cup \{o_\psi\},$ $T_{\psi_1} \wedge T(X_n) \wedge (\neg b_\psi \leftrightarrow (o_\psi \leftrightarrow o_{\psi_1} \wedge (X_n = X_{n-1}))))$, $F_\psi = F_{\psi_1}$	

Here $I_\psi = \bigcup_{i \le n} I_i \dot\cup O_i \dot\cup X_i$; $\mathit{init}(X_n)$ is the initial condition applied to copy X_n of the latches; and likewise $T(X_n)$ is the transition relation of C applied to the copy X_n. We use $X_n = X_{n-1}$ to denote the expression that all latches in X_n are equal to their counterparts in X_{n-1}. We omitted the construction for the conjunction and the Release operator due to the space limits. It is easy to verify that the transition relation is input-enabled and input-deterministic.

Proposition 3. *Given a circuit C and an alternation-free formula φ with k quantifiers, the size of the circuit C_φ is at most $|C| \cdot k + |\varphi|$.*

For each subformula ψ of φ, the output o_ψ in the circuit C_φ indicates that ψ must hold for the current computation path, and the latch x_ψ represent the requirements on the future of the computation that arise from the output o_ψ. The output b_ψ indicates that the requirements for subformula ψ are violated and a *bad state* is entered.

Proposition 4. *Let C be a circuit and let φ be an alternation-free HyperCTL* formula. $C \models \varphi$ holds iff the circuit C_φ admits a computation that shows output o_φ in the first step, that never outputs b_ψ for any of the subformulas ψ of φ, and that satisfies the fairness constraints.*

The proof of correctness proceeds again by structural induction on the structure of the formula. The search for paths of the form above can be performed by standard hardware model checkers.

					Verification time in s			
		Model	#Latches	#Gates	IC3	INT	BMC	
IF1	(NI1)	I2C Master	254	1207	95.17	1.13	0.07	×
IF2	(NI2)				53.08	1.16	0.08	×
IF3	(NI3)				168.96	1.38	-	✓
IF4	(NI4)				438.41	1.01	0.09	×
IF5	(NI5)				717.74	8.31	0.77	×
IF6	(NI6)				186.20	1.10	0.07	×
IF7	(NI7)				TO	6.82	0.55	×
IF8	(NI8)				1557.14	2.92	0.16	×
IF9	(NI2′)	Ethernet	21093	70837	TO	155.77	6.27	×
Sym1	(S1)	Bakery	46	1829	6.34	0.88	0.08	×
Sym2	(S2)				168.59	464.52	7.00	×
Sym3		Bakery.a	47	1588	69.12	TO	71.92	×
Sym4	(S3)	Bakery.a.n	47	1618	26.31	4.75	0.39	×
Sym5		Bakery.a.n.s	47	1532	66.41	TO	-	✓
Sym6	(S4)				16.83	TO	-	✓
Sym7	(S5)	Bakery.a.n.s.5proc	90	3762	97.45	TO	-	✓
Sym8	(S6)				13.59	TO	-	✓
Sym9	(S7)	Bakery.a.n.s.7proc	136	6775	312.53*	TO	-	✓
Huff1	(HD1)	Huffman_enc	19	571	3.08	37.19	-	✓
Huff2	(HD2)				0.62	0.09	0.02	×
8b10b_1	(HD1)	8b10b_enc	39	271	0.32	0.09	0.02	×
8b10b_2	(HD1′)				1.19	9.06	-	✓
8b10b_3	(HD2′)				0.03	0.04	0.02	×
8b10b_4	(HD1″)	8b10b_dec	19	157	0.05	0.09	-	✓
Hamm1	($HD1_1$)	Hamming_enc	27	47	0.02	0.04	0.02	×
Hamm2	($HD1_2$)				0.02	0.03	0.02	×
Hamm3	($HD1_3$)				0.03	0.04	0.02	×
Hamm3'	($HD1_3'$)				7.34	0.18	-	✓
Hamm4	($HD1_4$)				66.93	0.10	-	✓
Hamm5	($HD2_1$)				11.83	1.31	-	✓
Hamm6	($HD2_2$)				14.44	0.78	-	✓
Hamm7	(HD3)				12.23	1.25	-	✓

Table 1. Experimental results for the case studies.

5 Case Studies and Experimental Results

We have implemented the symbolic model checking approach from Section 4 as a transformation on Aiger circuits. We rely on standard hardware synthesis tools to compile VHDL and Verilog files into a circuit to which we apply our tool to obtain a new circuit. As the backend engine, we use the ABC model checker [11], which provides many of the modern verification algorithms, including IC3 [10]/PDR [18], interpolation (INT) [33], and SAT-based bounded model checking (BMC) [8]. All experiments ran on an Intel Core i5 processor (4278U) with 2.6 GHz. Table 1 shows the verification times for the circuits and properties

The tool and the experiments are available online [41].

considered in our case studies. We used the default settings of ABC in all runs, except the entry marked with *. The symbol ✓ indicates that an invariant was found, and $\times$ that a (counter)example was found.

The experiments show that our approach enables the verification of hyperproperties for hardware modules with hundreds or even thousands of latches. For finding counterexamples, bounded model checking was most effective, and for cases where an invariant was needed, the relative performance of IC3/PDR vs. interpolation was inconclusive. In addition to benchmarking, our goal for these case studies has been to explore the versatility of alternation-free HyperCTL* model-checking and the potential of our prototype tool. In the following subsections, we report on the setup and results of the case studies, as well as on the verification workflow from a user perspective. Our case studies come from three different areas: information flow, symmetry, and error resistant codes.

5.1 Case Study 1: Information Flow Properties of I2C

Our first case study investigates the information flow properties of an I2C bus master. I2C is a widely used bus protocol that connects multiple components in a master-slave topology. Even though the I2C bus has no security features, it has been used in security-critical applications, such as the smart cards of the German public health insurance, which led to exploits [44]. We analyzed a I2C bus master implementation from the open source repository `opencores.org`. A typical setup consists of one *master*, one *controller*, and several *slaves*. The master communicates to the slaves via two physical wires, the clock line (SCL) and the data line (SDA). The interface of the master towards the *controller* consists of 8 bit wide words for input and output of data, a 3-bit wide address to encode slave numbers, a system clock input, and several reset and control signals. We checked the I2C bus master implementation against the information flow properties shown in Table 2.

(NI1)	$\forall\pi.\forall\pi'.\ \square(\overline{\text{ADDR\_I}}_\pi = \overline{\text{ADDR\_I}}_{\pi'}) \Rightarrow \square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'})$
(NI2)	$\forall\pi.\forall\pi'.\ \square\,\overline{\text{DAT\_I}}_\pi = \overline{\text{DAT\_I}}_{\pi'} \Rightarrow \square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'})$
(NI3)	$\forall\pi.\forall\pi'.\ \square(\neg\text{WEn} \wedge \overline{\text{DAT\_I}}_\pi = \overline{\text{DAT\_I}}_{\pi'}) \Rightarrow \square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'})$
(NI4)	$\forall\pi.\forall\pi'.\ \square(\overline{\{\text{SDA\_I},\text{SCL\_I}\}}_\pi = \overline{\{\text{SDA\_I},\text{SCL\_I}\}}_{\pi'}) \Rightarrow \square(\text{DAT\_O}_\pi = \text{DAT\_O}_{\pi'})$
(NI5)	$\forall\pi.\ \square(\text{SDA\_Enable} \Rightarrow \mathcal{H}_{\{\text{SDA\_I},\text{SCL\_I}\},\{\text{DAT\_O}\}}\mathsf{false})$
(NI6)	$\forall\pi.\forall\pi'.\ \square(\overline{\text{SDA\_I}}_\pi = \overline{\text{SDA\_I}}_{\pi'}) \Rightarrow \square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'})$
(NI7)	$\forall\pi.\forall\pi'.\ \square(\overline{\text{DAT\_I}}_\pi = \overline{\text{DAT\_I}}_{\pi'}) \Rightarrow (\square(I_\pi = I_{\pi'}) \Rightarrow \diamondsuit\square(\text{SDA\_O}_\pi = \text{SDA\_O}_{\pi'}))$
(NI8)	$\forall\pi.\forall\pi'.\ \square(\overline{\{\text{SDA\_I},\text{SCL\_I}\}}_\pi = \overline{\{\text{SDA\_I},\text{SCL\_I}\}}_{\pi'}) \Rightarrow (\square(I_\pi = I_{\pi'}) \Rightarrow \diamondsuit\square(\text{DAT\_O}_\pi = \text{DAT\_O}_{\pi'}))$

Table 2. Information flow properties for the verification of the I2C bus master. In this list of properties, $P_\pi = P_{\pi'}$ is defined as $\bigwedge_{a\in P} a_\pi = a_{\pi'}$. $\overline{P}_\pi = \overline{P}_{\pi'}$ is defined as $(I \setminus P)_\pi = (I \setminus P)_{\pi'}$ where $P \subseteq \mathsf{AP}$ and $I \subseteq \mathsf{AP}$ are the inputs of the circuit.

From the controller to the bus. Property (NI1) states that there is no information flow with respect to the address to which the I2C master intends to send data,

and (NI2) with respect to the data words themselves. Both information flows are intended, and our tool reports the violation. We tried to bound the information flow between the first valuation of the 3 bit wide address input and the bus data by encoding [14] the quantitative information-flow property. While the information flow of 3 bit could be determined (QNI1), checking the upper bound of $\log 9 \approx 3.17$ bit (QNI2) led to a timeout. Property (NI3) states that when the *write enable* bit is not set, no information should flow from the controller inputs to the bus. This property is satisfied by the implementation.

From the bus to the controller. Property (NI4) claims the absence of information flow from the slaves to the controller, which is again legitimately violated by the implementation. Property (NI5) refines (NI4) to determine whether the flow can still happen when we only consider information received on SDA *while* the master sends data too. The branching time operator $\mathcal{H}$ in (NI5), called the Hide operator $\mathcal{H}_{I,O}\varphi$, is borrowed from the logic SecLTL [16] and expresses that information from the inputs I do not interfere with the outputs O. The Hide operator is easily expressible in HyperCTL* [14]. Property (NI5) is violated by the implementation, because the concurrent transmission of data on the bus by multiple masters can bring I2C into arbitration mode and changes the interpretation of information sent over the bus later.

Long-term information flow: Properties (NI7) and (NI8) claim that the information flows from (NI1) and (NI4) cannot happen for an arbitrary delay. These properties are violated, which indicates that information may not be eventually forgotten by the I2C master.

All properties on the I2C Master were easily analyzed by the model checker. In order to determine if our approach scales to even larger designs, we checked an adapted version of property (NI2) on an Ethernet IP core with 21093 latches. The counterexample was still found within seconds.

5.2 Case Study 2: Symmetry in Mutual Exclusion Protocols

In our second case study, we investigate symmetry properties of mutual exclusion protocols. Mutual exclusion is a classical problem in distributed systems, for which several solutions have been proposed and analyzed. Violation of symmetry indicates that some clients have an unfair advantage over the other clients.

Our case study is based on a Verilog implementation of the Bakery protocol [27] from the VIS verification benchmark. The Bakery protocol works as follows. When a process wants to access the critical section it draws a "ticket", i.e., it obtains a number that is incremented every time a ticket is drawn. If there is more than one process who wishes to enter the critical section, the process with the smallest ticket number goes first. When two processes draw tickets concurrently, they may receive tickets with the same number, so ties among processes with the same ticket must be resolved by a different mechanism, for example by comparing process IDs. The Verilog implementation has an input *select* to indicate the process ID that runs in the next step, and an input *pause* to indicate whether the step is stuttering. Each process n has a program counter $pc(n)$.

When process n is selected, the statement corresponding the program counter $pc(n)$ is executed. We are interested in the following HyperLTL property:

(S1)	$\forall\pi.\forall\pi'.\ \square(sym(select_\pi, select_{\pi'}) \ \wedge\ pause_\pi = pause_{\pi'}) \Rightarrow$ $\square(pc(0)_\pi = pc(1)_{\pi'} \ \wedge\ pc(1)_\pi = pc(0)_{\pi'})$

where $sym(select_\pi, select_{\pi'})$ means that process 0 is selected on path π when process 1 is selected on path π' and vice versa. Property (S1) states that, for every execution, there is another execution in which the *select* inputs corresponding to processes 0 and 1 are swapped and the outcome (i.e., the sequence of program counters of the processes) is also swapped. It is well known that it is impossible to accomplish mutual exclusion in an entirely symmetric fashion [32]. It is therefore not surprising that the implementation indeed violates Property (S1).

Inspecting the counterexample revealed, however, that the symmetry is broken even before the critical section is reached: if a non-existing process ID is selected by the variable *select*, process 0 proceeds instead. Property (S2) excludes paths on which a non-existing process ID is selected. The model-checker produced a counterexample in which processes 0 and 1 tried to access the critical section, but were treated differently.

(S2)	$\forall\pi.\forall\pi'.\ \square(\mathsf{sym}(select_\pi, select_{\pi'}) \ \wedge\ pause_\pi = pause_{\pi'} \ \wedge$ $select_\pi < 3 \wedge select_{\pi'} < 3) \Rightarrow$ $\square(pc(0)_\pi = pc(1)_{\pi'} \ \wedge\ pc(1)_\pi = pc(0)_{\pi'})$

Next, we parameterized the necessary symmetry breaking in the system. We introduced additional inputs indicating which process may move, in case of a tie of the tickets and extended the property by the assumption that the symmetry is broken symmetrically.

(S3)	$\forall\pi.\forall\pi'.\ \square(\mathsf{sym}(select_\pi, select_{\pi'}) \ \wedge\ pause_\pi = pause_{\pi'} \ \wedge$ $select_\pi < 3 \wedge select_{\pi'} < 3 \ \wedge\ \mathsf{sym}(sym\_break_\pi, sym\_break_{\pi'})) \Rightarrow$ $\square(pc(0)_\pi = pc(1)_{\pi'} \ \wedge\ pc(1)_\pi = pc(0)_{\pi'})$

Property (S3) is still violated by the implementation: the order in which the processes were checked depends on the process IDs and causes delays in how the program counters evolve. After contracting the comparison of process IDs into a single step, property (S3) became satisfied.

In further experiments, we changed the structure of property from the form (S3) $\forall\pi.\forall\pi'.\ \square\varphi \Rightarrow \square\psi$ to (S7) $\forall\pi.\forall\pi'.\ \psi\ \mathcal{W}\ \neg\varphi$, which removes the liveness part of the property, while maintaining the semantics (for input-deterministic and input-enabled systems). This change significantly reduced the verification times and enabled the verification of the protocol for up to 7 participants.

5.3 Case Study 3: Error Resistant Codes

Error resistant codes enable the transmission of data over noisy channels. While the correct operation of encoder and decoders is crucial for communication systems, the formal verification of their functional correctness has received little

attention. A typical model of errors bounds the number of flipped bits that may happen for a given code word length. Then, error correction coding schemes must guarantee that all code words have a minimal Hamming distance. Alternation-free HyperCTL* can specify that all code words produced by an encoder have a minimal Hamming distance of d:

$$\boxed{\text{(HDd)}} \quad \boxed{\forall\pi.\forall\pi'.\,\Diamond(\textstyle\bigvee_{a\in I} a_\pi \neq a_{\pi'}) \Rightarrow \neg\mathrm{Ham}_O(d-1,\pi,\pi')}$$

where I are the inputs denoting the data, O denote the code words, and the predicate $\mathrm{Ham}_O(d,\pi,\pi')$ is defined as $\mathrm{Ham}_O(-1,\pi,\pi') = \mathsf{false}$ and:

$$\mathrm{Ham}_O(d,\pi,\pi') = \big(\textstyle\bigwedge_{a\in O} a_\pi = a_{\pi'}\big)\ \mathcal{W}\ \big(\textstyle\bigvee_{a\in O} a_\pi \neq a_{\pi'}\ \wedge\ \bigcirc\mathrm{Ham}_O(d-1,\pi,\pi')\big).$$

We started with two simple encoders that are not intended to provide error resistance: a Huffman encoder from the VIS benchmarks, and an 8bit-10bit encoder from `opencores.org` that guarantees that the difference between the number of 1s and the number of 0s in the codeword is bounded by 2. As expected, encoders provide a Hamming distance of 1 (Huff1 and 8b10b_2), but not more (Huff2 and 8b10b_3). The experiments on these simple encoders were useful to determine the configuration of the command signals that enable the transmission of data. For example, checking the plain property as specified above for the 8bit-10bit encoder reveals that the reset signal must be set to false before sending data (8b10b_1). Similarly, for the 8bit-10bit *decoder*, we checked whether all codewords of Hamming distance 1 produce different outputs (8b10b_4).

Next, we considered an encoder for the 7-4-Hamming code, which encodes blocks of 4 bits into codewords of length 7, and guarantees a Hamming distance of 3. We started with finding out in which configuration the encoder actually sends encoded data (Hamm1 to Hamm4). With Hamm3 we discovered that the implementation deviates from the specification because the reset signal for the circuit is active high, instead of active low as specified. In Hamm3, we fixed the usage of the reset bit. We then scaled the specification to Hamming distances 2 and 3 (Hamm5 to Hamm7).

6 Conclusions

We presented a novel automata-based automatic technique to model-check HyperLTL and HyperCTL* specifications, and an implementation integrated with a state-of-the-art hardware model checker. Our case studies show that the implementation scales to realistic hardware designs; in one case we successfully checked a design with more than 20.000 latches. The logics HyperLTL and HyperCTL* proved to be versatile tools for the analysis of various kinds of properties.

Acknowledgements We thank Hans-Jörg Peter for valuable discussions and for synthesizing models for the case studies, Heinrich Ody for joint work on an early prototype of the tool, and Heidy Khlaaf for insightful comments on the paper.

References

1. Rajeev Alur, Pavol Černý, and Steve Zdancewic. Preserving secrecy under refinement. In *Proc. of ICALP'06*, volume 4052 of *LNCS*, pages 107–118. Springer, 2006.
2. Henrik Reif Andersen. A polyadic modal μ-calculus, 1994. Technical Report.
3. A. Askarov and A. Myers. A semantic framework for declassification and endorsement. In *Proc. ESOP*, pages 64–84. Springer, 2010.
4. Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic temporal logic for information flow security. In *Proc. of PLAS*. ACM, Jun. 2011.
5. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. of S&P*, pages 339–353. IEEE CS Press, 2008.
6. Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: asymmetric product programs for relational program verification. In *Proc. of LFCS'13*, volume 7734 of *LNCS*, pages 29–43. Springer, 2013.
7. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proc. CSFW*, pages 100–114, June 2004.
8. Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. Verifiying safety properties of a power PC microprocessor using symbolic model checking without BDDs. In *Proc. of CAV*, volume 1633 of *LNCS*, pages 60–71. Springer, 1999.
9. Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. `http://fmv.jku.at/hwmcc11/beyond1.pdf`, 2011. Accessed Feb 6, 2015. Via website: `http://fmv.jku.at/aiger/`.
10. Aaron R. Bradley. SAT-based model checking without unrolling. In *Proc. of VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
11. Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Proc. of CAV*, volume 6174 of *LNCS*, pages 24–40. Springer, 2010.
12. Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. of DAC'90*, pages 46–51. IEEE CS Press, 1990.
13. Koen Claessen, Niklas Eén, and Baruch Sterin. A circuit approach to LTL model checking. In *Proc. of FMCAD*, pages 53–60, 2013.
14. Michael Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proc. of POST*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.
15. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proc. IEEE Symp. on Computer Security Foundations*, pages 51–65, June 2008.
16. Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Proc. of VMCAI*, volume 7148 of *LNCS*, pages 169–185. Springer, 2012.
17. Deepak D'Souza, Raveendra Holla, K. R. Raghavendra, and Barbara Sprick. Model-checking trace-based information flow properties. *Journal of Computer Security*, 19(1):101–138, 2011.
18. Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *Proc. of FMCAD*, pages 125–134, 2011.
19. Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, 1995.

20. Bernd Finkbeiner and Markus N. Rabe. The linear-hyper-branching spectrum of temporal logics. *it - Information Technology*, 56:273–279, November 2014.
21. Peter Gammie and Ron van der Meyden. MCK: Model checking the logic of knowledge. In *Proc. of CAV*, volume 3114 of *LNCS*, pages 479–483. Springer, 2004.
22. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, 1982.
23. Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE CS Press, 1982.
24. Marieke Huisman, Patrik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *Proc. of CSFW*. IEEE CS Press, 2006.
25. Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, pages 1–16, 1998.
26. Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM TOCL*, 2(3):408–429, 2001.
27. Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974.
28. Martin Lange and Étienne Lozes. Model-checking the higher-dimensional modal mu-calculus. In *Proc. of FICS*, volume 77 of *EPTCS*, pages 39–46, 2012.
29. Alessio Lomuscio, Charles Pecheur, and Franco Raimondi. Automatic verification of knowledge and time with NuSMV. In *Proc. of IJCAI*, pages 1384–1389, 2007.
30. Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Proc. of CAV*, volume 5643 of *LNCS*, pages 682–688. Springer, 2009.
31. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
32. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
33. Kenneth L. McMillan. Craig interpolation and reachability analysis. In *Proc. of SAS*, volume 2694 of *LNCS*, page 336. Springer, 2003.
34. Artur Meski, Wojciech Penczek, Maciej Szreter, Bozena Wozna-Szczesniak, and Andrzej Zbrzezny. Bounded model checking for knowledge and linear time. In *Proc. of AAMAS*, pages 1447–1448. IFAAMAS, 2012.
35. Dimiter Milushev. *Reasoning about Hyperproperties*. PhD thesis, Katholieke Universiteit Leuven, Faculty of Engineering, Celestijnenlaan 200A, box 2402, B3001 Heverlee, Belgium, 6 2013.
36. Dimiter Milushev and Dave Clarke. Towards incrementalization of holistic hyperproperties. In *Proc. of POST*, volume 7215 of *LNCS*, pages 329–348. Springer, 2012.
37. Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32:321–330, 1984.
38. David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proc. of LICS*, pages 422–427. IEEE CS Press, 1988.
39. Martin Otto. Bisimulation-invariant PTIME and higher-dimensional μ-calculus. *Theoretical Computer Science*, 224:237–265, 1998.
40. Wojciech Penczek and Alessio Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. In *Proc. of AAMAS*, pages 209–216. IFAAMAS, 2003.

41. Markus N. Rabe. MCHyper: A model checker for hyperproperties. `http://www.react.uni-saarland.de/tools/mchyper/`, 2015. Accessed Feb 6, 2015.
42. Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *Proc. of ISSS*, volume 3233 of *LNCS*, pages 174–191. Springer, 2004.
43. A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.
44. Wolfgang Thielke. Code geknackt. Link to article in media archive: `http://www.focus.de/finanzen/news/krankenkassen-code-geknackt_aid_148829.html`, 1994. Accessed Feb 6, 2015.
45. Ron van der Meyden and Chenyi Zhang. Algorithmic verification of noninterference properties. *Electr. Notes Theor. Comput. Sci.*, 168:61–75, 2007.
46. Moshe Y. Vardi. Alternating automata and program verification. In *Computer Science Today*, volume 1000 of *LNCS*, pages 471–485. Springer, 1995.
47. Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS'86*, pages 332–344. IEEE CS Press, 1986.
48. Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.