Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Master's Thesis

Verifying Hyperliveness

Author:
Norine Coenen

Advisor:
Prof. Bernd Finkbeiner, Ph.D.

Reviewers:
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr. Cas Cremers

Submitted: 30th September 2019

Preface

This Master's Thesis is based on the paper "Verifying Hyperliveness" [15] by Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. This paper was accepted at the 31st International Conference on Computer-Aided Verification (CAV'19) held from July 13 to July 18 2019 in New York City, NY, USA. According to the CORE2018 ranking [50], this conference is an A* conference. It accepts 'Regular Papers' of up to 16 pages (plus references), as well as 'Tool Papers' and 'Industrial Experience Reports and Case Studies' (up to 8 pages plus references each). This year, there were 258 submissions and 67 papers were accepted. Thus, the overall acceptance rate was 26% and, according to the business meeting, 23% for regular papers (52 were accepted).

Our paper is a regular paper that successfully went through the full double blind review process. The relevant artifacts were submitted to the artifact evaluation committee for evaluation. This was done on invitation to obtain the artifact evaluation seal, but it was not required for publication.

Norine Coenen is the corresponding author for this paper and managed the publication process of this paper with Springer. Further, Norine Coenen presented this paper at the conference in New York City.

This thesis is based on the accepted paper which presents a new approach to the verification of hyperliveness properties expressed as HyperLTL formulas with quantifier alternation. Strategic choice is substituted for existential choice, thereby expanding the set of problems to which automatic verification can be applied. The paper mainly considers the model checking problem and later explores the problem of automatically synthesizing reactive systems.

In the following, the individual contributions of each co-author to the paper are clarified. Then, the technical content follows, based on the model checking parts of the paper. The presentation of the content is adapted, especially the presentation of the implementation and the experimental evaluation is much more detailed than in the paper. Finally, this thesis includes all the reviews that we have received for our paper and the submitted artifacts.

Individual Contributions

In the following, the individual contributions are clarified.

- Norine Coenen did the majority of the work leading to the model checking part of the paper (Section 3.1, Subsection “Hardware Model Checking with Given Strategies” in Section 5). She developed the problem and the approach to the presented solution in discussions with Bernd Finkbeiner and César Sánchez. The majority of the research was done by Norine Coenen. She also implemented the extension of MCHyper, performed the corresponding experimental evaluation and wrote the sections of the paper on model checking (including Subsection “Model Checking HyperLTL” in Section 2). Further, she contributed large parts to the rest of the paper and provided critical feedback to the sections on synthesis after having discussed the presented research with Leander Tentrup.
- Leander Tentrup mainly contributed to the synthesis part of this paper (Section 3.2, Section 4, Subsection “Strategy and System Synthesis” in Section 5). In cooperation with Norine Coenen, he developed the solution to the synthesis problem for liveness hyperproperties. He implemented the extension of BoSy, performed the corresponding experiments and wrote the sections of the paper regarding the synthesis problem. He also contributed to writing the rest of the paper.
- Bernd Finkbeiner supervised the project. He discussed the approach and the research regarding the model checking problem with Norine Coenen and provided critical feedback to earlier versions of the paper. He also contributed to writing parts of the paper.
- César Sánchez participated in discussions about the model checking problem and its solution resulting in the presented approach. He also contributed to writing parts of the paper.

All authors approved the final version of the paper. For her Master’s Thesis, Norine Coenen would like to submit the model checking part of the paper “Verifying Hyperliveness”.

Saarbrücken, 30th September 2019

Norine Coenen Bernd Finkbeiner

César Sánchez Leander Tentrup

Statement and Declaration of Consent

I hereby confirm that I have written the parts of the paper as described in the “Individual Contributions” section and that I have not used any other media or materials than the ones referred to in this thesis.

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Erklärung und Einverständniserklärung

Ich erkläre hiermit, dass ich die beschriebenen Teile des Papiers selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Saarbrücken, 30th September 2019

Norine Coenen

Contents

Abstract

Hyperliveness describes the class of liveness hyperproperties. Just like for trace properties, that are expressible, e.g., in Linear-time Temporal Logic (LTL), *liveness* captures that *something good eventually has to happen. Hyperproperties* are a generalization of trace properties where relations between multiple execution traces can be expressed. This is necessary to formalize information-flow policies like generalized noninterference or the symmetry requirement. Hyperproperties can be expressed in the temporal logic HyperLTL which extends LTL by explicit quantification over trace variables.

Verifying hyperproperties has been studied before. In particular, there is an automata-based algorithm for model checking HyperLTL. Unfortunately, its complexity increases exponentially with every quantifier alternation in the formula. Therefore, the only practical tool for model checking HyperLTL, MCHyper, is limited to alternation-free formulas. The alternation-free fragment, however, cannot express hyperliveness properties since these hyperproperties require, in general, a quantifier alternation in the formula.

We present a proof technique for model checking HyperLTL that avoids the exponential increase in complexity by shifting the perspective to a *game-theoretic view.* In our approach, we consider the model-checking problem as a game between the $\forall$-player and the $\exists$-player. The moves of the $\exists$-player are determined by an appropriate strategy, thereby fixing the existentially quantified traces.

We have implemented this approach as an extension of the tool MCHyper such that it now can handle formulas with up to one quantifier alternation. Our experimental evaluation shows the practical applicability of the extended version of MCHyper by examining symmetry in mutual exclusion algorithms and checking the cleanness of emission control modules of cars.

Chapter 1

Introduction

Computer systems are getting more and more complex and, at the same time, are being used in more and more areas of our lives. Therefore, these systems must work as intended, especially in safety-critical areas like air traffic control [33, 59]. Formally verifying these computer systems against their specifications results in mathematical guarantees on their correctness.

When verifying the correctness of a system, its specification, describing the desired system behavior in natural language, has to be formalized. One way to do this is to express the specification mathematically as a logical formula. Temporal logics, like the Linear-time Temporal Logic (LTL) [48], are used to characterize the allowed system behavior. Such an LTL characterization describes a trace property that decides for every individual execution trace whether it is permitted in the system or not.

Unfortunately, checking individual execution traces does not suffice to capture all possible errors in a system [46]. A prominent example for this are the Meltdown and Spectre attacks on computer processors [35, 39]. These processors have been proven correct with respect to their trace properties [46], yet, they were still vulnerable to the so-called *side-channel attacks*. In these attacks, timing differences in multiple system executions are used to deduce possibly secret information in the computer's memory. This leakage of secret information violates the intended *information-flow policies* for the computer processors.

Information-flow policies are an example of a class of system requirements that cannot be captured by trace properties. Consider, for example, *noninterference* [30], an important information-flow policy on systems with secret and public values. Intuitively, noninterference states that all execution traces with the same public inputs should also have the same public outputs, regardless of the values of the secret inputs. To show that some system violates this requirement, we have to find two system traces with the same public inputs but different public outputs. Only checking individual execution traces cannot reveal such a violation. Instead, multiple execution traces need to be considered and compared to one another. This is not possible with trace properties, e.g., when using LTL. In fact, noninterference is an example for a hyperproperty [12]. Hyperproperties are temporal properties that relate multiple execution traces.

HyperLTL [13] is a temporal logic that can express hyperproperties. It extends LTL by explicit quantification over trace variables which allows specifying relations between several previously quantified traces. The ability to relate multiple execution traces makes hyperproperties more expressive than trace properties. This increased expressiveness is necessary to formalize, for example, information-flow policies. Noninterference between a secret input h and a public output o can be specified in HyperLTL by the following formula:

$$\forall \pi.\ \forall \pi'.\ \Box\Big(\bigwedge_{i \in I \setminus \{h\}} i_\pi \leftrightarrow i_{\pi'} \Big) \rightarrow \Box(o_\pi \leftrightarrow o_{\pi'}).$$

This formula states that all pairs of traces π and π', where the public inputs $i \in I \setminus \{h\}$ are the same in every step, must also have the same public outputs o in every step. Thus, noninterference ensures that a change in the secret input h alone cannot cause a change in the public output o.

To express noninterference for non-deterministic systems, the hyperproperty *generalized noninterference* [42] has to be considered. The following HyperLTL formula specifies generalized noninterference:

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

The existential quantifier is needed to handle the non-determinism in the system. This specification allows the system to behave non-deterministically on the public variables without permitting information flowing from secret inputs into public outputs.

Verification of hyperproperties is conceptually more complicated than the verification of trace properties because the specified relations between multiple execution traces have to be considered. For model checking HyperLTL formulas without quantifier alternation, the self-composition [5] of the system can be used. This self-composed system contains one system copy for every quantifier. To check an alternation-free HyperLTL formula, it then suffices to check a trace property (expressible in, e.g., LTL) on an appropriate self-composition of the system. Thus, the complexity of model checking the alternation-free fragment of HyperLTL is the same as model checking LTL. It is NLOGSPACE-complete in the size of the system and PSPACE-complete in the size of the formula.

However, the alternation-free fragment cannot express all formulas of interest. Generalized noninterference, for example, has a quantifier alternation in the formula. In general, many hyperliveness properties do not fall into the alternation-free fragment.

Noninterference, on the other hand, belongs to the alternation-free fragment of HyperLTL. Moreover, it is an example of a safety hyperproperty. Hypersafety properties are characterized as having bad prefixes that cannot be extended in any way to satisfy the safety hyperproperty. Generalized noninterference, on the other hand, is a liveness hyperproperty [12]. Hyperliveness properties do not have bad prefixes, i.e., any prefix violating the liveness hyperproperty can be extended such that the hyperliveness property is satisfied. For generalized noninterference, for example, we have to add an appropriate trace π'' for each offending pair of traces π, π'.

Other examples of hyperliveness properties include symmetry and robust cleanness. *Symmetry* in mutual exclusion algorithms [23] ensures that no process has an unfair advantage. *Robust cleanness* [17] of software systems rules out unexpected jumps between the output behavior of several traces when there has not been a significant difference in the input sequences.

Expressing liveness hyperproperties like generalized noninterference, symmetry, and robust cleanness in HyperLTL requires, in general, a quantifier alternation in the formula. For HyperLTL formulas with quantifier alternation, the model checking problem is more complex.

There is an automata-based model checking algorithm for the full logic of HyperLTL [23]. In this algorithm, $\forall$ quantifiers are represented as negated $\exists$ quantifiers over the negated subformula. Thus, the non-deterministic Büchi automaton built by the algorithm has to be complemented. Complementation of non-deterministic Büchi automata, however, is exponential in the size of the automaton [36]. Therefore, the complexity of the model-checking problem of HyperLTL depends on the number of quantifier alternations in the formula. Already for HyperLTL formulas with one quantifier alternation, the model checking problem is PSPACE-complete in the size of the system and EXPSPACE-complete in the size of the formula.

This exponential increase in complexity prevents the efficient implementation of this algorithm. Instead, the only practical model checking tool for HyperLTL, MCHyper, exploits the self-composition technique and is, therefore, limited to the alternation-free fragment of HyperLTL. This, however, excludes hyperproperties like generalized noninterference, symmetry, and robust cleanness from automatic verification.

In this thesis, we present a proof technique that uses strategies to avoid this exponential increase in complexity. We consider the model-checking problem as a game between a ∀-player and an ∃-player. Depending on the choices of the ∀-player, the strategy for the ∃-player determines the existentially quantified traces. If the strategy for the ∃-player always makes the correct choices for the existentially quantified traces, then the system satisfies the hyperproperty that is model checked. Such a strategy is called *winning*, and it generates the witnesses for the existential trace quantifiers.

There are, however, cases where no appropriate strategy for the ∃-player exists, even though the system satisfies the hyperproperty. We explore this lack in completeness in more detail and show how *prophecy variables* [1] can be used to partly overcome the incompleteness.

We implemented this proof technique as an extension of the model checker MCHyper such that the tool can handle HyperLTL formulas with up to one quantifier alternation. To demonstrate the practical usefulness of this extension, we evaluated it on two sets of examples from the literature that considered symmetry and robust cleanness [17, 23]. Direct model checking of these hyperliveness properties was not possible with the previous version of MCHyper and, therefore, the hyperproperties had to be approximated by alternation-free HyperLTL formulas. With our extension, we can model check the hyperproperties of interest directly, and we report our verification results for these two sets of experiments.

1.1 Related Work

There are many temporal logics that are used to express system specifications. Besides the linear-time logic LTL, there are also branching-time temporal logics like CTL [11] and CTL* [18]. Even though these logics support existential and universal quantification over the computation paths in a tree, they cannot express hyperproperties: As soon as a second path is quantified, the previously quantified path is no longer accessible, and the subformula cannot refer to this path anymore.

To express branching-time hyperproperties, we need to be able to access several paths at the same time to establish relations between them. The temporal logic HyperCTL* [13] can do this. It extends CTL* with quantification over path variables, similar to the extension of LTL to HyperLTL. Just like CTL* subsumes LTL, HyperCTL* subsumes HyperLTL [20].

There are several other logics, besides HyperLTL and HyperCTL*, that can express hyperproperties including first-order logic extended with the equal-level predicate E (FO[<,E]) [22]. These logics differ in their expressive power and the complexity of their satisfiability problem [14].

The automata-based model checking algorithm for HyperLTL also works for HyperCTL* [23]. Also, in this case, the complexity depends on the number of quantifier alternations in the formula.

The practical model checking tool MCHyper [23] can handle alternation-free HyperLTL formulas. We extend MCHyper to obtain the first practical model checking tool for HyperLTL that can handle formulas with up to one quantifier alternation.

In MCHyper, self-composition [5] is used to reduce the problem of checking a hyperproperty to checking a trace property. This technique has been used before, for example for the verification of information-flow policies [6, 32, 55, 56]. However, in these approaches, the verification method is specialized to the information-flow policy that is considered. Our approach is much more flexible as it allows the practical verification of every hyperproperty that is expressible as a HyperLTL formula with up to one quantifier alternation.

In our extension of MCHyper, we use strategies to determine the existential traces incrementally. This is only an approximation of the HyperLTL semantics. In the case of a $\forall^*\exists^*$ HyperLTL formula, the choice of the existentially quantified traces depends on the infinite traces chosen for the universally quantified trace variables. So in our game-theoretic view, the $\exists$-player has infinite lookahead. Previous work on finite lookahead [34] can, therefore, not be directly applied to our setting.

To capture the HyperLTL semantics more accurately, we use prophecy variables [1]. These variables predict the future behavior of the $\forall$-player and make this information accessible to the strategy. Prophecy variables were previously used to obtain simulations between automata [40] as well as in the verification of branching-time properties expressed as CTL* formulas [16].

Besides the model checking problem, also various other problems of HyperLTL have been considered before. The satisfiability problem has been explored [19, 24, 26] where, given a HyperLTL formula, a model satisfying this formula is found if one exists. In runtime verification, the monitoring problem of HyperLTL has been studied [25, 28, 31]. When a system is monitored, an alarm is raised as soon as a violation of the specification is detected. The model-checking problem for quantitative hyperproperties has been considered as well [29]. This class of hyperproperties can be model checked by exploiting model-counting algorithms.

Moreover, the synthesis problem of HyperLTL has been studied. In synthesis, we try to automatically find an implementation that satisfies a given specification. For HyperLTL, a synthesis algorithm for the alternation-free fragment exists [27]. This algorithm performs bounded synthesis [21] to ensure that the smallest possible solution is found.

Recently, this algorithm has been extended to work for HyperLTL formulas with up to one quantifier alternation [15]. This extension is also based on a game-theoretic view. The extended synthesis algorithm searches for a system implementation and an additional system representing a strategy. This strategy is used to determine the existential choices and shows that the generated system satisfies the hyperproperty. When the system implementation is given, synthesis can be used to model check this implementation [15]. In this case, the synthesis algorithm only tries to find the system representing a winning strategy. For sufficiently small strategies, strategy synthesis can be used to automate the manual work of finding a winning strategy in our approach.

Chapter 2

Preliminaries

We start by giving some basic definitions. First, we introduce the temporal logic LTL before defining its extension to hyperproperties, HyperLTL. We then specify transition systems as our model of computation. After explaining Büchi automata, we recap the automata-based model checking algorithm for HyperLTL. Preparing our shift in perspective to a game-theoretic view, we formally introduce strategies.

2.1 LTL

Linear-time Temporal Logic (LTL) [48] is a temporal logic that can express trace properties. Using LTL, the correct behavior of execution traces of a system can be defined by describing the allowed sequences of states.

Let AP be a finite set of atomic propositions and $\Sigma = 2^{\mathrm{AP}}$ the corresponding alphabet. A *trace* $t \in \Sigma^\omega$ is an infinite sequence of elements of Σ. With $t[i]$, we denote the i-th position in trace t.

LTL formulas are built according to the following grammar, where $a \in \mathrm{AP}$:

$$\psi ::= a \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi.$$

In addition to the boolean operators *negation* ($\neg\psi$) and *conjunction* ($\varphi \wedge \psi$), we allow the derived boolean connectives (*disjunction*: $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$, *implication*: $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$, *equivalence*: $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$). Additionally, we use the temporal operators *next* ($\bigcirc \psi$ states that ψ should hold in the next state) and *until* ($\varphi \,\mathcal{U}\, \psi$ states that φ should be *true* in every state until ψ holds) as well as the derived temporal operators (*release*: $\varphi \,\mathcal{R}\, \psi \equiv \neg(\neg\varphi \,\mathcal{U}\, \neg\psi)$, *eventually*: $\Diamond \varphi \equiv \mathit{true} \;\mathcal{U}\; \varphi$, *globally*: $\Box \varphi \equiv \neg \Diamond \neg\varphi$, as well as *weak until*: $\varphi \,\mathcal{W}\, \psi \equiv \Box \varphi \vee (\varphi \;\mathcal{U}\; \psi)$).

For a trace t and a natural number i representing the current point in time, the semantics of LTL is given by the satisfaction relation $\vDash$ that is defined as follows:

$$\begin{array}{lll}
t, i \vDash a & \text{iff} & a \in t[i] \\
t, i \vDash \neg\psi & \text{iff} & t, i \nvDash \psi \\
t, i \vDash \varphi \wedge \psi & \text{iff} & t, i \vDash \varphi \text{ and } t, i \vDash \psi \\
t, i \vDash \bigcirc\psi & \text{iff} & t, i+1 \vDash \psi \\
t, i \vDash \varphi\,\mathcal{U}\,\psi & \text{iff} & \exists j \geq i.\ t, j \vDash \psi \text{ and } \forall i \leq k < j.\ t, k \vDash \varphi.
\end{array}$$

We write $t \vDash \varphi$ for $t, 0 \vDash \varphi$.

Every trace property is the intersection of a safety and a liveness trace property [3]. Safety properties [2, 38] stipulate that *nothing bad ever happens.* Liveness properties [3, 38], on the other hand, stipulate that *something good eventually happens.*

2.2 HyperLTL

The temporal logic HyperLTL [13] extends LTL by adding explicit quantification over trace variables π from an infinite set of trace variables $\mathcal{V}$. By linking atomic propositions to these trace variables, it is possible to define relations between several traces, thereby expressing hyperproperties.

HyperLTL formulas are built according to the following grammar, where $a \in \mathrm{AP}$ and $\pi \in \mathcal{V}$:

$$\begin{aligned}
\psi &::= \forall\pi.\,\psi \mid \exists\pi.\,\psi \mid \varphi, \text{ and} \\
\varphi &::= a_\pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi.
\end{aligned}$$

The atomic proposition a is linked to the trace that is referenced by π by writing a_π. Like for LTL, we derive the boolean connectives $(\vee, \rightarrow, \leftrightarrow)$ as well as the additional temporal operators $(\mathcal{R}, \Diamond, \Box, \mathcal{W})$.

Let $Tr \subseteq \Sigma^\omega$ be a set of traces and $\Pi : \mathcal{V} \rightarrow \Sigma^\omega$ an assignment that maps trace variables to infinite traces. We denote the empty assignment with $\{\}$. A given trace assignment Π can be updated by $\Pi[\pi \mapsto t]$ such that the resulting trace assignment is the same as Π, except that the trace variable π maps to trace t.

The semantics of HyperLTL is given by the satisfaction relation $\vDash_{Tr}$ defined as follows:

$$\begin{array}{lll}
\Pi, i \vDash_{Tr} a_\pi & \text{iff} & a \in \Pi(\pi)[i] \\
\Pi, i \vDash_{Tr} \neg\varphi & \text{iff} & \Pi, i \nvDash_{Tr} \varphi \\
\Pi, i \vDash_{Tr} \varphi \wedge \psi & \text{iff} & \Pi, i \vDash_{Tr} \varphi \text{ and } \Pi, i \vDash_{Tr} \psi \\
\Pi, i \vDash_{Tr} \bigcirc\varphi & \text{iff} & \Pi, i+1 \vDash_{Tr} \varphi \\
\Pi, i \vDash_{Tr} \varphi \mathcal{U} \psi & \text{iff} & \exists j \geq i.\ \Pi, j \vDash_{Tr} \psi \text{ and } \forall i \leq k < j.\ \Pi, k \vDash_{Tr} \varphi \\
\Pi, i \vDash_{Tr} \forall\pi.\, \psi & \text{iff} & \text{for all } t \in Tr \text{ it holds that } \Pi[\pi \mapsto t], i \vDash_{Tr} \psi \\
\Pi, i \vDash_{Tr} \exists\pi.\, \psi & \text{iff} & \text{there is some } t \in Tr \text{ such that } \Pi[\pi \mapsto t], i \vDash_{Tr} \psi.
\end{array}$$

We write $Tr \vDash \varphi$ for $\{\}, 0 \vDash_{Tr} \varphi$.

Consider a HyperLTL formula of the form $\mathcal{Q}_1\pi_1 \cdots \mathcal{Q}_n\pi_n.\ \varphi$, where, for all $i \leq n$ we have that $\mathcal{Q}_i \in \{\forall, \exists\}$, and φ is a quantifier free HyperLTL formula that denotes the *body of the formula.* If all quantifiers in the quantifier prefix are of the same kind, we call the HyperLTL formula alternation free, or equivalently, we say the formula is in the alternation-free fragment of HyperLTL. If it only contains $\forall$ quantifiers (or $\exists$ quantifiers), we say it is a universal formula (it is an existential formula, respectively). Further, we say a HyperLTL formula has one quantifier alternation if the quantifier prefix consists of a sequence of $\forall$ quantifiers followed by a sequence for $\exists$ quantifiers (e.g., $\forall\pi.\ \exists\pi'.\ \varphi$), or vice versa (e.g., $\exists\pi.\ \forall\pi'.\ \varphi$).

Hyperproperties can, similar to trace properties, be classified as safety and liveness hyperproperties [12]. These hyperproperties are called hypersafety and hyperliveness properties, respectively. Hypersafety properties stipulate that *nothing bad ever happens.* They are characterized as having bad prefixes, i.e., there are finite sets of finite traces that cannot be extended in a way that the resulting (possibly infinite) set of infinite traces satisfies the safety hyperproperty.

Analogously, hyperliveness properties stipulate that *something good eventually happens.* These hyperproperties are characterized as not having any bad prefixes. This means that every finite set of finite traces can be extended to a (possibly infinite) set of infinite traces that satisfies the liveness hyperproperty. Following this definition, every hyperproperty with $\exists$ quantifiers is a hyperliveness property since it does not have any bad prefixes. Instead, every violation of the hyperproperty by a set of traces can be overcome by adding the appropriate traces for the $\exists$ quantifiers. Moreover, every hyperproperty is an intersection of a hypersafety and a hyperliveness property [12].

2.3 Transition Systems

Our model of computation are transition systems that model reactive systems by reading input sequences and producing output sequences. Formally, transition systems consume sequences over an input alphabet by transforming their internal state in every step and producing the corresponding sequence over an output alphabet.

The input alphabet is $\Upsilon = 2^I$, where I is the finite set of input propositions. Analogously, $\Gamma = 2^O$ is the output alphabet with the finite set of output propositions O. A Γ-labeled Υ-*transition system* $\mathcal{S}$ is a tuple $\langle S, s_0, \tau, l\rangle$, where S is a finite set of states, $s_0 \in S$ is the designated initial state, $\tau \colon S \times \Upsilon \to S$ is the transition function, and $l \colon S \to \Gamma$ is the state-labeling function.

The transition function τ is generalized to work with sequences over Υ. The generalized transition function is $\tau^* \colon \Upsilon^* \to S$. It is recursively defined by $\tau^*(\epsilon) = s_0$ and $\tau^*(\upsilon_0 \cdots \upsilon_{n-1}\upsilon_n) = \tau(\tau^*(\upsilon_0 \cdots \upsilon_{n-1}), \upsilon_n)$, for every $\upsilon_0 \cdots \upsilon_{n-1}\upsilon_n \in \Upsilon^+$.

A *trace* ρ through a Γ-labeled Υ-transition system is determined by the infinite input sequence $\upsilon = \upsilon_0\upsilon_1 \ldots \in \Upsilon^\omega$. The transition system reacts to this input sequence by producing an infinite output sequence $\gamma = \gamma_0\gamma_1\gamma_2 \ldots \in \Gamma^\omega$, where, for every $i \geq 0$, $\gamma_i = l(\tau^*(\upsilon_0 \ldots \upsilon_{i-1}))$. The trace ρ is then defined as $(\upsilon_0 \cup \gamma_0)(\upsilon_1 \cup \gamma_1) \ldots \in \Sigma^\omega$ with $AP = I \cup O$. We denote the set of traces of $\mathcal{S}$ by *traces*$(\mathcal{S})$.

We define the product $\mathcal{S} \times \mathcal{S}'$ of two transition systems $\mathcal{S} = \langle S, s_0, \tau, l\rangle$ and $\mathcal{S}' = \langle S', s_0', \tau', l'\rangle$ as $\mathcal{S} \times \mathcal{S}' = \langle S \times S', (s_0, s_0'), \tau'', l''\rangle$. $\mathcal{S} \times \mathcal{S}'$ is a Γ^2-labeled Υ^2-transition system with $\tau''((s, s'), (\upsilon, \upsilon')) = (\tau(s, \upsilon), \tau'(s', \upsilon'))$ and $l''((s, s')) = (l(s), l'(s'))$.

A transition system $\mathcal{S}$ satisfies an LTL formula ψ if, and only if, $t \vDash \psi$ for all traces $t \in$ *traces*$(\mathcal{S})$. $\mathcal{S}$ satisfies a HyperLTL formula ψ if, and only if, *traces*$(\mathcal{S}) \vDash \psi$.

2.4 Büchi Automata

A Büchi automaton [10] is an automaton over infinite words that, given such a word, either accepts or rejects this word. Thus, a Büchi automaton defines a language $\mathcal{L}(\mathcal{A})$ containing all words the automaton accepts.

Formally, a Büchi automaton $\mathcal{A}$ over a finite alphabet Σ is a tuple $\langle Q, q_0, \delta, F\rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta \colon Q \times \Sigma \to 2^Q$

is the transition function, and $F \subseteq Q$ is the set of accepting states. A safety automaton is a Büchi automaton where every state is accepting.

The transition function δ returns, for a state $q \in Q$ and some $a \in \Sigma$, the set of possible successor states Q'. If all of these sets only have one element, i.e., the successor state is unique, we call the automaton *deterministic*. Otherwise, it is *non-deterministic* and may choose to move to any of the successor states in Q'.

A run of a non-deterministic Büchi automaton $\mathcal{A}$ on a word $\rho \in \Sigma^\omega$ is an infinite sequence of states $s_0, s_1, \ldots \in Q^\omega$ such that $s_0 = q_0$ and $\delta(s_i, \rho[i]) = s_{i+1}$ for all $i \geq 0$. A run is *accepting* if, and only if, there are infinitely many positions i such that $s_i \in F$, i.e., infinitely many accepting states are visited.

The set of words ρ that have an accepting run on $\mathcal{A}$ is the language of $\mathcal{A}$. Formally, the language of $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{\rho \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } \rho\}$. An automaton $\mathcal{A}$ accepts a transition system $\mathcal{S}$, written $\mathcal{S} \vDash \mathcal{A}$, if $\mathit{traces}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{A})$.

2.5 Model Checking HyperLTL

In model checking, we want to show that a given system satisfies a given specification. The automata-based model checking algorithm for HyperLTL [23] can handle the full logic. However, for every quantifier alternation in the HyperLTL formula, the complexity of the algorithm increases exponentially.

In the simplest case, when considering alternation-free formulas, the complexity of model checking HyperLTL is the same as the complexity of model checking LTL formulas [51]. Thus, the model-checking problem of the alternation-free fragment of HyperLTL is PSPACE-complete in the size of the formula and NLOGSPACE-complete in the size of the transition system. Already with one quantifier alternation in the formula, the model checking problem is in EXPSPACE in the size of the formula and in PSPACE in the size of the transition system.

For alternation-free HyperLTL formulas, there is the practical model checking tool MCHyper [23]. It uses the idea of self-composition [5], where several system copies are combined into a new system. Over this new system, we then can check a standard trace property.

Let a tuple $\vec{v} \in \Upsilon^n$ contain n elements of Υ. We can apply a function $\tau \colon \Upsilon \to \Gamma$ to a tuple $\vec{v} \in \Upsilon^n$ to obtain the tuple where the function has been applied to every component in $\vec{v}$: $\tau \circ \vec{v} = (\tau(\vec{v}[1]), \ldots, \tau(\vec{v}[n])) \in \Gamma^n$. The function *zip* maps an n-tuple of sequences to a single sequence of n-tuples, for example, $\mathit{zip}([1,2,3],[4,5,6]) = [(1,4),(2,5),(3,6)]$.

We define the n-fold self-composition of a transition system $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ as $\mathcal{S}^n = \langle S^n, \vec{s_0}, \tau_n, l_n \rangle$, where $\vec{s_0} \coloneqq (s_0, \ldots, s_0) \in S^n$, $\tau_n(\vec{s}, \vec{v}) \coloneqq \tau \circ \mathit{zip}(\vec{s}, \vec{v})$, and $l_n(\vec{s}) \coloneqq l \circ \vec{s}$ for every $\vec{s} \in S^n$ and $\vec{v} \in \Upsilon^n$. The set of traces of the n-fold self composition of $\mathcal{S}$ is then the set $\mathit{traces}(\mathcal{S}^n)$ that is defined as $\{\mathit{zip}(\rho_1, \ldots, \rho_n) \mid \rho_1, \ldots, \rho_n \in \mathit{traces}(\mathcal{S})\}$. Intuitively, a trace through the self-composed system $\mathcal{S}^n$ is a tuple of n traces through $\mathcal{S}$, one trace through every system copy.

The self-composed system is used to model check alternation-free formulas. The following theorem justifies this application of the self-composition for alternation-free HyperLTL formulas. The notation $\mathit{zip}(\varphi, \pi_1, \pi_2, \ldots, \pi_n)$ for some HyperLTL formula φ is used to combine all the trace variables $\pi_1, \pi_2, \ldots, \pi_n$ (occurring free in φ) into a fresh trace variable π^*.

Theorem 2.1 (Self-composition for alternation-free formulas [23]) Let $\mathcal{Q} = \forall$ for the universal fragment and $\mathcal{Q} = \exists$ for the existential fragment. For a transition system $\mathcal{S}$ and a HyperLTL formula of the form $\mathcal{Q}\pi_1.\mathcal{Q}\pi_2.\ldots\mathcal{Q}\pi_n.\ \varphi$ it holds that $\mathcal{S} \vDash \mathcal{Q}\pi_1.\mathcal{Q}\pi_2.\ldots\mathcal{Q}\pi_n.\ \varphi$ if, and only if, $\mathcal{S}^n \vDash \mathcal{Q}\pi^*.\ \mathit{zip}(\varphi, \pi_1, \pi_2, \ldots, \pi_n)$.

For HyperLTL formulas with quantifier alternation, the self-composition cannot be used in the same way: The system copies corresponding to a $\forall$ quantifier would have to be treated differently from the ones corresponding to an $\exists$ quantifier. Therefore, the model checker MCHyper is limited to the alternation-free fragment of HyperLTL.

Model checking HyperLTL formulas with quantifier alternation was, so far, only possible using the theoretical automata-based algorithm [23]. This algorithm involves the complementation of a non-deterministic Büchi automaton, once per quantifier alternation in the formula. Complementing such an automaton is exponential in the size of the automaton [36].

Interesting HyperLTL formulas like symmetry or robust cleanness involve a quantifier alternation. Nevertheless, we would like to model check these hyperproperties automatically. This can be done by fixing the existential choice and strengthening the formula to the universal fragment [17, 23]. This strengthened formula can then be model checked automatically using MCHyper. However, it is not guaranteed that the strengthened formula is correct, i.e., that it implies the original formula. The correctness has to be shown manually since the problem of checking implications becomes undecidable [19].

In this thesis, we present a proof technique for model checking HyperLTL formulas with quantifier alternation that circumvents this complexity problem while still giving strong correctness guarantees. We implement this technique as an extension of MCHyper that can handle HyperLTL formulas with up to one quantifier alternation. Our technique uses strategies to determine the choice for the existentially quantified traces.

2.6 Strategies

Strategies are used in game theory to determine the moves of the individual players [47]. We use strategies to fix the choices for the existentially quantified traces when considering the model checking problem for HyperLTL as a game between the $\forall$-player and the $\exists$-player. For a $\forall\pi.\ \exists\pi'.\ \varphi$ HyperLTL formula, the strategy for the $\exists$-player observes the choices of the $\forall$-player for the universally quantified traces and determines the corresponding existentially quantified traces.

A strategy $\sigma\colon \Upsilon^* \to \Upsilon$ maps finite histories over Υ to a value in Υ. We use transition systems to represent strategies. A transition system $\mathcal{S} = \langle S, s_0, \tau, l\rangle$ computes a strategy σ if $\sigma(\vec{v}) = l(\tau^*(\vec{v}))$ for every $\vec{v} \in \Upsilon^*$. If there exists such a transition system computing σ, we say that strategy σ is *finite-state*.

We use finite-state strategies to determine the inputs of transition systems that represent existentially quantified system copies. Let $\mathcal{S} = \langle S, s_0, \tau, l\rangle$ be such a transition system over input alphabet Υ and output alphabet Γ. Further, let $\mathcal{S}_\sigma = \langle S', s'_0, \tau', l'\rangle$ be the transition system computing the finite-state strategy $\sigma\colon (\Upsilon')^* \to \Upsilon$. We define the transition system $\mathcal{S} \mid\mid \sigma = \mathcal{S} \mid\mid \mathcal{S}_\sigma$ as $\langle S \times S', (s_0, s'_0), \tau^{||}, l^{||}\rangle$, where $\tau^{||}\colon (S \times S') \times \Upsilon' \to (S \times S')$ is the transition function and defined as $\tau^{||}((s, s'), \upsilon') = (\tau(s, l'(s')), \tau'(s', \upsilon'))$, and the state-labeling function $l^{||}\colon (S \times S') \to \Gamma$ is defined as $l^{||}(s, s') = l(s)$ for every $s \in S$, $s' \in S'$, and $\upsilon' \in \Upsilon'$.

We now show how to use strategies to model check HyperLTL formulas with quantifier alternation while avoiding the increase in complexity inherent to the automata-based algorithm.

Chapter 3

Model Checking HyperLTL with Quantifier Alternations

In this chapter, we present our approach to model checking HyperLTL formulas with quantifier alternation. Our main idea is a shift in perspective to a *game-theoretic view* where we consider the model checking problem as a game between the $\forall$-player and the $\exists$-player. To simplify the presentation, we consider formulas of the form $\forall^*\exists^*\varphi$ and $\exists^*\forall^*\varphi$, where φ is quantifier-free. In our approach, we use strategies to determine the choices of the $\exists$-player. Thus, we are substituting strategic choice for existential choice.

3.1 Using Strategies

The strategies we use to determine the choices of the $\exists$-player may depend on the choices made by the $\forall$-player. This reflects the dependencies between the quantifiers that are introduced by a quantifier alternation in the formula. In the case of a $\forall^*\exists^*$ HyperLTL formula, the correct choice of the $\exists$-player for the existentially quantified traces depends on the system traces chosen by the $\forall$-player for the universally quantified trace variables.

For an $\exists^*\forall^*$ HyperLTL formula, the choice of the $\exists$-player for the existentially quantified traces does not depend on the choices of the $\forall$-player. Instead, one choice for the existentially quantified traces has to work for all possible choices for the universally quantified traces. Therefore, the strategy does not depend on the choices of the $\forall$-player. The strategy instead encodes the correct witnesses for the existentially quantified traces.

We say that the strategy for the $\exists$-player is *winning* if it picks the correct witnesses for the existential traces for every possible behavior of the $\forall$-player.

If the $\exists$-player has a winning strategy, then the considered HyperLTL formula holds on the system because the strategy determines the required witnesses for the existential trace quantifiers.

We can use a given strategy for the $\exists$-player to reduce the model checking problem with quantifier alternation to model checking of an alternation-free formula. The following theorem shows how the strategy is used to eliminate the $\exists$ quantifiers.

Theorem 3.1 (Substituting Strategic for Existential Choice)
Let $\mathcal{S}$ be a transition system over input alphabet Υ.
It holds that $\mathcal{S} \vDash \forall \pi_1. \forall \pi_2 \dots \forall \pi_n. \exists \pi'_1. \exists \pi'_2 \dots \exists \pi'_m. \varphi$ if there is a strategy $\sigma : (\Upsilon^n)^* \to \Upsilon^m$ that determines the existentially quantified traces such that $\mathcal{S}^n \times (\mathcal{S}^m \mid\mid \sigma) \vDash \forall \pi^*. \mathit{zip}(\varphi, \pi_1, \pi_2, \dots \pi_n, \pi'_1, \pi'_2, \dots, \pi'_m)$.
It holds that $\mathcal{S} \vDash \exists \pi_1. \exists \pi_2 \dots \exists \pi_m. \forall \pi'_1. \forall \pi'_2 \dots \forall \pi'_n. \varphi$ if there is a strategy $\sigma : (\Upsilon^0)^* \to \Upsilon^m$ that determines the existentially quantified traces such that $(\mathcal{S}^m \mid\mid \sigma) \times \mathcal{S}^n \vDash \forall \pi^*. \mathit{zip}(\varphi, \pi_1, \pi_2, \dots \pi_m, \pi'_1, \pi'_2, \dots, \pi'_n)$.

The proof of this theorem essentially uses the traces chosen by the strategy as witnesses for the existentially quantified traces.

Proof Let σ be a strategy that determines the existentially quantified traces, then we define a witness for the existential trace quantifiers $\exists \pi'_1. \exists \pi'_2 \dots \exists \pi'_m$ as the sequence of inputs $\upsilon = \upsilon_0 \upsilon_1 \dots \in (\Upsilon^m)^\omega$ such that, for every $i \geq 0$ and every $\upsilon'_i \in \Upsilon^n$, $\upsilon_i = \sigma(\upsilon'_0 \upsilon'_1 \dots \upsilon'_{i-1})$; analogously, we define a witness for the existential trace quantifiers $\exists \pi_1. \exists \pi_2 \dots \exists \pi_m$ as the sequence of inputs $\upsilon = \upsilon_0 \upsilon_1 \dots \in (\Upsilon^m)^\omega$ such that $\upsilon_i = \sigma(\upsilon'_0 \upsilon'_1 \dots \upsilon'_{i-1})$ for every $i \geq 0$ and every $\upsilon'_i \in \Upsilon^0$. $\square$

Using the given strategy to determine the choices of the $\exists$-player reduces the model checking problem to checking an alternation-free formula. As discussed in Section 2.5, this model checking problem is exponentially simpler than the one with a quantifier alternation. This gain in efficiency is especially significant if the given strategy is sufficiently small.

Corollary 3.2 (Model Checking with Given Strategies)
The model-checking problem for HyperLTL formulas with one quantifier alternation and a given strategy for the existential quantifiers is in PSPACE in the size of the formula and NLOGSPACE in the size of the product of the strategy and the system.

This technique is similar to Skolemization [54] where functions replace existentially quantified variables in first-order logic. These functions depend on the variables that were quantified in the formula before. Similar to the way Skolemization works for arbitrary quantifier structures, our approach can also be extended to more than one quantifier alternation. One strategy is needed for every block of existential quantifiers, and these strategies may only observe the traces that were previously quantified in the formula. To simplify the presentation, this thesis focusses on formulas with one quantifier alternation.

If we can find a winning strategy for the ∃-player, we can use this to show that the original HyperLTL property holds on the system by using Theorem 3.1. However, such a strategy does not always exist. Some systems satisfy a HyperLTL formula with quantifier alternation, but there is no winning strategy for the ∃-player. This is because strategies only observe the finite prefix of the universally quantified traces chosen by the ∀-player and have no knowledge about the infinite future of these traces.

Consider, for example, a system with arbitrary sequences of a and $\neg a$ and the following HyperLTL formula:

$$\forall \pi.\ \exists \pi'.\ \bigcirc a_\pi \leftrightarrow a_{\pi'} \tag{3.1}$$

This formula asks the ∃-player to predict the move of the ∀-player in the second step. Even though this system satisfies the HyperLTL formula, there is no winning strategy for the ∃-player. Every possible strategy picks some value for a on π' in the first step. Seeing this value for $a_{\pi'}$, the ∀-player can choose to set a_π to the opposite truth value in the second step, thereby falsifying the formula.

The problem here is that the ∃-player needs information about the future behavior of the ∀-player to make the correct decision. In the following, we show how this problem can be overcome by making the necessary information about the future accessible to the strategy of the ∃-player.

3.2 Using Prophecy Variables

The information about the future that is needed in the strategy to make the right choice can be made accessible by using prophecy variables [1]. These are variables that are used to predict the future behavior of the ∀-player.

We assume that the prophecy variables always predict the future correctly. In this case, the strategy can use the values of the prophecy variables to make its decisions. We only require our original hyperproperty to hold if the prophecy variables indeed predicted the future correctly. If the prophecy variables predict do not predict the future correctly, we do not require the original hyperproperty to hold. To reflect this interpretation, we modify the hyperproperty that we want to model check by adding the correctness of the predictions of the prophecy variables as a premise.

Consider again the example from above where we have a system producing arbitrary sequences of a and $\neg a$ and Formula 3.1. In this case, the value of a on π' in the first step depends on the value of a on π in the second step. To pick the correct value for $a_{\pi'}$ in the first step, the strategy, therefore, needs information about the value of a_π in the second step.

We introduce a prophecy variable p that predicts the value of a_π in the second step, i.e., $p \leftrightarrow \bigcirc a_\pi$. The strategy can then use the value of p to make its decision for the value of $a_{\pi'}$ in the first step. The winning strategy for the $\exists$-player sets $a_{\pi'}$ in the first step to the same value as the prophecy variable p, i.e., $a_{\pi'} \leftrightarrow p$.

If the prediction of the prophecy variable p was correct, then the strategy ensures that indeed $\bigcirc a_\pi \leftrightarrow a_{\pi'}$ and the HyperLTL formula is satisfied. To reflect this assumption on the correctness of the prophecy variable's prediction, we modify the HyperLTL formula that we want to check, resulting in the following formula:

$$\forall \pi.\ \exists \pi'.\ (p_\pi \leftrightarrow \bigcirc a_\pi) \ \rightarrow\ (\bigcirc a_\pi \leftrightarrow a_{\pi'}). \tag{3.2}$$

The prophecy variables are added as inputs to the system that are controlled by the environment and don't affect the system's behavior. Formally, we add a set P of prophecy variables to a Γ-labeled Υ-transition system $\mathcal{S} = \langle S, s_0, \tau, l\rangle$ by defining the Γ-labeled $(\Upsilon \cup P)$-transition system $\mathcal{S}^P = \langle S, s_0, \tau^P, l\rangle$ including the inputs P where $\tau^P \colon S \times (\Upsilon \cup P) \to S$. For all $s \in S$ and $\upsilon^P \in \Upsilon \cup P$, $\tau^P(s, \upsilon^P) = \tau(s, \upsilon)$ for $\upsilon \in \Upsilon$ obtained by removing the variables in P from υ^P, i.e., $\upsilon^P =_{\setminus P} \upsilon$.

For a single prophecy variable p, the following theorem shows the proof technique for model checking with prophecy variables:

Theorem 3.3 (Model checking with Prophecy Variables)
For a transition system $\mathcal{S}$ and a HyperLTL formula of the form $\forall^*\exists^*\varphi$, let p be a fresh atomic proposition and let ψ be a quantifier-free HyperLTL formula over the universally quantified trace variables $\pi_1, \pi_2 \dots \pi_n$ that captures the information about the future that p should predict. It holds that $\mathcal{S} \vDash \forall \pi_1.\ \forall \pi_2 \dots \forall \pi_n.\ \exists \pi'_1.\ \exists \pi'_2 \dots \exists \pi'_m.\ \varphi$ if, and only if, for system $\mathcal{S}^{\{p\}}$ with new input p, $\mathcal{S}^{\{p\}} \vDash \forall \pi_1.\ \forall \pi_2 \dots \forall \pi_n.\ \exists \pi'_1.\ \exists \pi'_2 \dots \exists \pi'_m.\ \Box(p_{\pi_1} \leftrightarrow \psi) \to \varphi$.

The proof of this theorem exploits that the prophecy variable is a fresh atomic proposition.

Proof It is easy to see that the original specification implies the modified specification, since the φ is the conclusion of the implication.

Assume that the modified specification holds. Since the prophecy variable p is a fresh atomic proposition, and ψ does not refer to the existentially quantified trace variables, we can, for every choice of the universally quantified traces, always choose the values of p such that it predicts the future correctly, i.e., that p is true whenever ψ holds. In this case, the conclusion φ and therefore the original formula must hold. □

It is important that the prophecy variables only predict the future of the universally quantified traces. The prophecy formula ψ should not refer to any existentially quantified trace variables. If this was allowed, the ∃-player could falsify the assumption, i.e., break the premise of the implication, instead of trying to satisfy the original property.

To see this, consider again our example from above. If p would depend on $a_{\pi'}$ ($\psi = a_{\pi'}$), then we would obtain the following modified formula:

$$\forall \pi.\ \exists \pi'.\ (p_\pi \leftrightarrow a_{\pi'}) \ \rightarrow\ (\bigcirc a_\pi \leftrightarrow a_{\pi'}).$$

By assigning $a_{\pi'}$ to the opposite truth value of p_π, the strategy can satisfy this formula by simply falsifying the assumption that the prediction of the prophecy variable was correct.

We are only interested in strategies that try to satisfy the original formula. We do not want to allow the strategy to influence the truth value of the premise. Doing this would lead to a proof method that was not sound. To ensure that this is not possible, the prophecy variables may only refer to universally quantified trace variables.

In the case of an $\exists^*\forall^*$ HyperLTL formula, no prophecy variables are needed since the existentially quantified traces do not depend on the choice for the universally quantified traces.

Even though prophecy variables help us overcome the problem with future dependencies, they are not enough to obtain a complete proof technique. There are still cases where, even though the system satisfies the HyperLTL formula, we cannot prove this with the presented technique. This is because we cannot always find appropriate prophecy variables that allow us to define a winning strategy. To see this, consider a system with arbitrary sequences of a and b together with the following specification:

$$\begin{aligned}\forall \pi.\ \exists \pi'.\ & b_{\pi'} \wedge \square(b_{\pi'} \leftrightarrow \bigcirc \neg b_{\pi'}) \\ & \wedge (\ \ a_{\pi'} \rightarrow (a_\pi \ \mathcal{W}\ (\ \ b_{\pi'} \wedge \neg a_\pi))) \\ & \wedge (\neg a_{\pi'} \rightarrow (a_\pi \ \mathcal{W}\ (\neg b_{\pi'} \wedge \neg a_\pi)))\end{aligned}$$

This formula requires the existentially quantified trace to alternate between b and $\neg b$, thereby marking even and odd positions in the trace. Moreover, trace π' intuitively has to predict whether π will switch from a to $\neg a$ for the first time at an even or at an odd position. To obtain a winning strategy that proves that our system satisfies this HyperLTL formula, we need a prophecy variable that predicts precisely this.

Since the even and odd positions are captured by $b_{\pi'}$, we would like to use this atomic proposition in the prophecy formula ψ. This, however, is not possible since the trace variable π' is existentially quantified. As explained above, the $\exists$-player could use this to violate the premise of the formula on purpose, so she does not have to fulfill the original property.

If we had $\psi = a_\pi \,\mathcal{W}\, (b_{\pi'} \wedge \neg a_\pi)$ in our example, the $\exists$-player can violate the premise $p_\pi \leftrightarrow (a_\pi \,\mathcal{W}\, (b_{\pi'} \wedge \neg a_\pi))$ by violating her specification that b and $\neg b$ have to alternate. This is, for example, the case in the following scenario. The $\forall$-player picks the trace $\{a\}\{a\}\{\}^\omega$ for π and the prophecy p_π predicts that at the first position where $\neg a_\pi$ holds, $b_{\pi'}$ will hold as well. Assume the $\exists$-player behaves according to her specification, i.e., considering only the atomic proposition b, the trace $(\{b\}\{\})^\omega$ is picked for π'. In this case, the prediction of the prophecy variable p is correct, i.e., in the first position where we have $\neg a_\pi$ we also have $b_{\pi'}$.

Assume now that the $\exists$-player violates her specification and instead picks, for example, the trace $\{b\}\{\}^\omega$ where only $\{\}$ is repeated infinitely often. Then the prediction of the prophecy variable p suddenly is incorrect since now at the first position where $\neg a_\pi$ holds, $b_{\pi'}$ is not *true* anymore.

The incorrect prediction of the prophecy variable means that the premise of the modified formula is violated. Therefore, the violation of the $\exists$-player's specification does not lead to the hyperproperty being violated. Instead, this violation is the reason for the hyperproperty to be true since the premise is violated. Nevertheless, we cannot conclude that the original hyperproperty holds. This shows again that this proof method is unsound when the prophecy formula refers to existentially quantified trace variables.

We, thus, cannot use any formula for ψ that includes existentially quantified trace variables. At the same time, no other atomic proposition in our formula or system marks even and odd positions such that we could use it instead of $b_{\pi'}$. Without such an atomic proposition, we are not able to express counting properties in HyperLTL since, just like LTL, HyperLTL can only express non-counting properties [43]. Due to this inherent restriction, our proof method cannot be complete for HyperLTL. In the following, we will explore how we need to extend HyperLTL to obtain a complete proof method.

3.3 Towards Completeness

As stated above, HyperLTL, as well as LTL, cannot express counting properties. When LTL is extended with quantification over fresh atomic propositions ($\exists q.\ \varphi$), the logic QPTL [53] with the following grammar is obtained:

$$\psi ::= a \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc\psi \mid \psi\,\mathcal{U}\,\psi \mid \exists q.\ \psi.$$

For the propositional quantification, the semantics is defined as follows:

$$t, i \vDash \exists q.\ \psi \quad \text{iff} \quad \exists t' \in 2^{\mathrm{AP}\cup\{q\}}.\ t' =_{\setminus\{q\}} t \text{ and } t', i \vDash \psi$$

There has to be some valuation of the fresh atomic proposition q on trace t resulting in trace t' that satisfies ψ.

QPTL can express counting properties [52] by quantifying over new atomic propositions that mark, for example, the even and the odd positions on the trace.

We use a similar extension for HyperLTL to obtain a logic that can express counting properties:

$$\begin{aligned}\psi &::= \forall\pi.\,\psi \mid \exists\pi.\,\psi \mid \varphi, \text{ and}\\ \varphi &::= a_\pi \mid \neg\varphi \mid \varphi\wedge\varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \exists q.\ \varphi.\end{aligned}$$

Note that this logic is not HyperQPTL [14, 49] as the propositional quantifiers are not necessarily in the quantifier prefix. It is, however, possible to transform formulas from this logic into HyperQPTL formulas by pulling the propositional quantifiers to the front, just like QPTL formulas can be brought into the prenex normal form [52, 53]. Formulas of the extended logic can be verified with the standard model checking algorithm for temporal hyperlogics [13, 49].

Quantification over additional atomic propositions allows us to express arbitrary ω-regular languages. This increase in expressiveness makes it possible to find prophecy formulas ψ that encode counting properties.

In the following, we show that our proof technique using strategies and prophecy variables is complete for this extended logic. In this setting, completeness means that if a formula holds on a system, it is possible to prove this Theorems 3.3 and 3.1.

Our completeness result is restricted to formulas of the form $\forall^*\exists^*\varphi$, where φ is a quantifier-free formula describing a safety property. Note that this does not mean that the formulas represent hypersafety properties. For example, the hyperliveness properties discussed in the introduction, generalized noninterference and robust cleanness, both belong to this fragment.

Theorem 3.4 (Completeness Result)
For a finite-state transition system $\mathcal{S}$ and a quantifier-free formula φ describing a safety property, if $\mathcal{S} \models \forall \pi_1. \forall \pi_2 \ldots \forall \pi_n. \exists \pi'_1. \exists \pi'_2 \ldots \exists \pi'_m. \varphi$, then there exists a set P of atomic propositions used as prophecy variables, y fresh atomic propositions $q_1 \ldots q_y$, and a set $\{\psi_p \mid p \in P\}$ of quantifier-free formulas over the universally quantified trace variables $\pi_1, \pi_2 \ldots \pi_n$ and the propositions $q_1 \ldots q_y$, such that, for system $\mathcal{S}^P$ with additional input variables P, $\mathcal{S}^P \models \forall \pi_1. \forall \pi_2 \ldots \forall \pi_n. \exists \pi'_1. \exists \pi'_2 \ldots \exists \pi'_m. \exists q_1 \ldots q_y. \ \square(\bigwedge_{p \in P} \ p_{\pi_1} \leftrightarrow \psi_p) \rightarrow \varphi$ can be proven with the technique from Theorem 3.1.

The proof gives a construction to find the required prophecy formulas ψ_p.

Proof (Sketch) The main idea of the proof is to build non-deterministic safety automata that accept the universally quantified traces whenever there are corresponding existentially quantified traces such that all traces together satisfy the safety property φ. We then introduce prophecy variables p that predict whether the future behavior of the universally quantified traces is in the languages of these automata. The strategy uses the values of the prophecy variables to decide whether taking a particular action is safe. By always taking safe actions, the strategy ensures that the safety property φ is satisfied. $\square$

The full proof of Theorem 3.4 can be found in Appendix A. The extension of this result to the case where the underlying LTL formula φ is a liveness property is left for future work.

Consider the model-checking problem of LTL again. In our model-checking technique, the user has to find the strategy and the appropriate prophecy formulas. This manual effort is, in many cases, easy for the user to perform since she already has a good understanding of the system and why it satisfies the hyperproperty.

In our practical experiments, the incompleteness of our technique for model checking HyperLTL was never a problem. In many cases, it is not even necessary to add prophecy variables because the considered hyperproperties do not involve future dependencies. The presented proof technique is, thus, practically useful despite the incompleteness result.

Chapter 4

Implementation and Experimental Evaluation

We have integrated the model checking technique with a manually provided strategy into the HyperLTL hardware model checker MCHyper. In the following, we describe this implementation and report on our experimental results. All experiments reported in this chapter ran on a machine with dual-core Core i7, 3.3 GHz, and 16 GB memory.

4.1 Implementation of MCHyper

The model checker MCHyper [23] is a hardware model checker for the alternation-free fragment of HyperLTL. We have extended this tool to handle formulas with up to one quantifier alternation. Before describing our extension of MCHyper, we describe how the original version of the tool works.

The core of MCHyper is implemented in the functional programming language Haskell. A frontend written in Python handles the file management and the interface to the backend tool ABC [9].

MCHyper is a hardware model checker that takes as inputs the system circuit C that should be model checked and the alternation-free HyperLTL formula $\forall\pi_1 \dots \forall\pi_n.\ \varphi$ (or $\exists\pi_1 \dots \exists\pi_n.\ \varphi$) that should hold on this system. The hardware circuit C representing the system has boolean variables and can perform arbitrary boolean operations on these variables. The time advances synchronously whenever the clock of the circuit ticks and it can keep and update its system state that is stored in its so-called *latches*.

```
 1  aag 3 2 0 1 1
 2  2
 3  4
 4  7
 5  6 3 5
 6  i0 in1
 7  i1 in2
 8  o0 out
 9  c
10  or-gate
```

Figure 4.1: Description of an OR-gate in the AIGER format.

Hardware circuits can be represented as *And-Inverter-Graphs* encoded in the AIGER format [7] which MCHYPER uses as its input format for the system. And-Inverter-Graphs only contain AND-gates and negations. Nevertheless, they can perform any boolean operation since conjunction and negation together form an operator base and all other operators can be expressed by using only conjunction and negation. For example, disjunction can be defined as $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$.

The AIGER format encodes an And-Inverter-Graph by assigning a variable index i to every boolean variable used in the graph. When a variable with the variable index i is used in the AIGER circuit, it appears as $2i$, when the negation of this variable is used then $2i + 1$ is used in the AIGER representation. Note that the first variable has variable index 1, so it occurs as `2`, or in the negated version as `3` in the AIGER circuit, and `0` and `1` are reserved for *false* and *true*, respectively.

An example of the description of an And-Inverter-Graph in the AIGER format is shown in 4.1. It starts with a header (line 1) giving the highest used variable index, the number of inputs, the number of latches, the number of outputs, and the number of AND-gates. This is followed by the lists of input variables (lines 2 and 3), latches, and output variables (line 4) which are specified before the AND-gates are defined (line 5). Then, a symbol table (lines 6 to 8) can assign names to inputs, latches, and outputs. For example, `i0 in1` in line 6 assigns the name `in1` to the first input in the input list (identified by `i0`). An optional comment (following a line only containing the letter `c`) may describe the circuit behavior (lines 9 and 10).

The AIGER circuit in Figure 4.1 defines an OR-gate. The header (line 1) states that it uses 3 variables (indexes $1, 2$ and 3), has 2 inputs, 0 latches, 1 output, and 1 AND-gate. Variables 1 and 2 are defined as inputs in lines 2 and 3 by giving the corresponding circuit representation of these variables ($2i$ for the variable with index i). The AND-gate in line 5 computes the con-

```
module or (input in1, input in2, output out)
    assign out = !(!in1 && !in2)
endmodule
```

Figure 4.2: Description of an Or-gate in Verilog.

junction of the two negated inputs (`3` for the negation of variable 1 and `5` for the negation of variable 2) and stores the result in variable 3 (represented as `6`). In line 4, the negation of variable 3 is defined as the output of the circuit. This precisely reflects the definition of the disjunction above.

When the system is specified in a hardware description language like Verilog, it is possible to translate this representation into the Aiger format using tools like Yosys [58]. Verilog is a powerful hardware description language that is used in industry. It allows specifying so-called `modules`, making the code more modular and reusable. An individual module has inputs and outputs, and its description defines the values assigned to the outputs based on the inputs. These definitions can use arbitrary boolean operations or define and use functions. Using a clock input, it is also possible to specify the system behavior over time. The synthesizable part of Verilog can then be automatically translated into the register transfer level (RTL) representation and from there into a netlist of gates representing the actual hardware.

Figure 4.2 shows a Verilog module that defines an Or-gate. This can be translated into an And-Inverter-Graph represented in the Aiger format using the tool Yosys. In fact, Yosys generates the Aiger circuit shown in Figure 4.1.

With the system and the HyperLTL formula as inputs, MCHyper builds a new Aiger circuit that can be given to the backend tool abc to check whether the hyperproperty holds. The idea behind this new circuit is that it contains the system and the formula, and has outputs that reflect whether the system violates the formula or not.

Figure 4.3 shows how MCHyper works for HyperLTL formulas without quantifier alternation. Given the input system C and an alternation-free formula with n quantifiers, the created circuit contains n copies of the system C, one for every quantifier in the formula. As a result, one trace through the resulting circuit C^n is a tuple of n traces through the system C. This technique is called self-composition [5] and it is a standard technique to reduce the verification of hyperproperties to the verification of a trace property over the appropriate self-composition of the system [6, 55].

Moreover, the new Aiger circuit contains a circuit C_φ that encodes the Büchi automaton for the body φ of the input formula. The construction

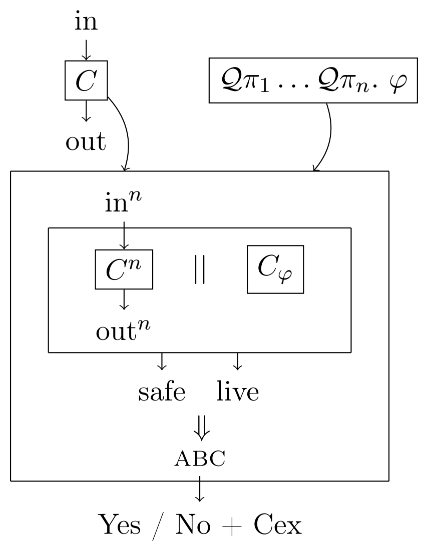

Figure 4.3: Model checking HyperLTL formulas from the alternation-free fragment (where $\mathcal{Q} \in \{\forall, \exists\}$) with MCHyper using self-composition.

used to build this automaton follows the standard translation from LTL to alternating automata [45, 57]. An equivalent non-deterministic automaton can be constructed using the standard algorithm by Miyano and Hayashi [44]. The resulting automaton accepts a pair of traces whenever these two traces together satisfy the alternation-free HyperLTL formula. To check whether the system satisfies the input formula, the automaton circuit C_φ is composed with the self-composed system C^n so that the automaton can observe the chosen system traces and either accept or reject them.

Since every hyperproperty is an intersection of a safety and liveness part [12], we consider these two parts of our input formula separately. The circuit that MCHyper builds has two outputs, one stating that the safety part of the hyperproperty is violated and one stating that the liveness part is violated. The safety check monitors the traces for a bad prefix that violates the safety part. The liveness check is performed via a reduction to a safety check using lassos to represent infinite traces [8]. A lasso describes a finite prefix that leads to some system state that is stored. The rest of the lasso then describes a path through the system starting and ending in this stored state. This describes a loop that can be taken infinitely often and on which the liveness condition has to be satisfied. Lassos are a standard representation for infinite traces through finite-state systems. The output corresponding to a liveness violation, therefore, states that a lasso in the system was found that did not satisfy the liveness requirements of the input formula.

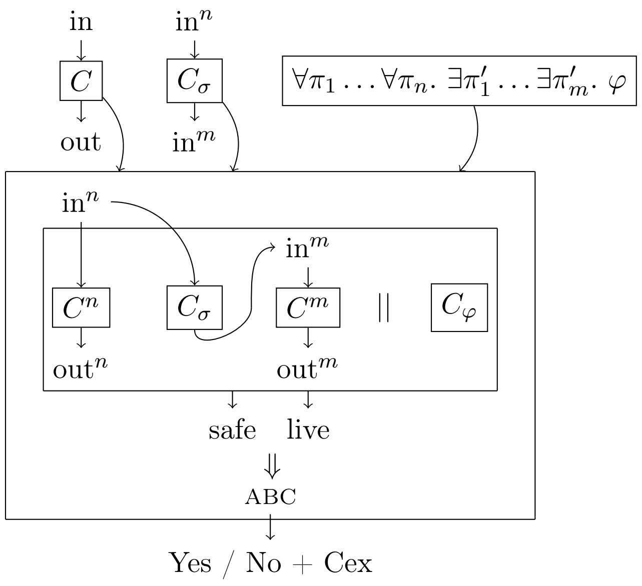

Figure 4.4: Model checking $\forall^n\exists^m$ HyperLTL with MCHYPER using self-composition and a given strategy.

The resulting AIGER circuit exposes the inputs of all system copies and the outputs indicating a safety or a liveness violation This circuit is then given to the backend tool ABC, which is used as a reachability checker. ABC tries to find inputs to the system copies that lead to a safety or a liveness violation and thereby show that the desired hyperproperty does not hold. If ABC does not manage to do so, then the hyperproperty holds on the system and MCHYPER reports this result to the user. If, on the other hand, ABC finds such an input sequence, the corresponding values are stored, and this counterexample is given to the user.

4.2 Extension of MCHYPER

We have extended MCHYPER to work with HyperLTL formulas of up to one quantifier alternation. The main difference to the previous version is that there now is a new input C_σ, a circuit encoding a strategy that fixes the choice for the existential traces. This is used in the self-composition to combine the universal with the existential system copies.

Figure 4.4 shows the extended version of MCHYPER for $\forall^n\exists^m$ HyperLTL, Figure 4.5 shows it for $\exists^n\forall^m$ HyperLTL. In both cases, the strategy is encoded as an AIGER circuit. Its inputs are the inputs to the universal system

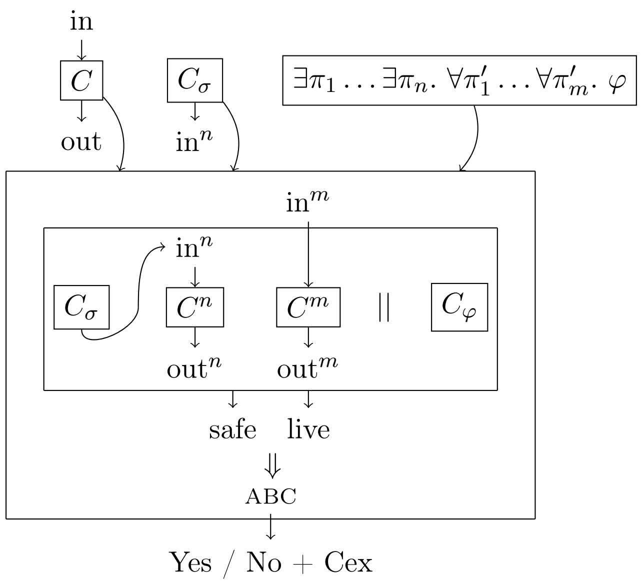

Figure 4.5: Model checking $\exists^n\forall^m$ HyperLTL with MCHyper using self-composition and a given strategy.

copies corresponding to the universal quantifiers preceding the existential quantifiers in the formula.

For $\forall^n\exists^m$ HyperLTL (Figure 4.4), the inputs to the strategy are n times the inputs to the system C. For $\exists^n\forall^m$ HyperLTL (Figure 4.5), there are no universal quantifiers before the existential ones, so the strategy has no inputs and, instead, encodes the n existential traces that are the witnesses. This ensures that the strategy has all the information about the universal system copies on which it is allowed to depend. Based on this information, the strategy determines the inputs to the existential system copies, thereby choosing the existential traces. The inputs to the existential system copies are the outputs of the strategy. For a system with a single input `in` and a HyperLTL formula of the form $\forall\pi_1.\ \forall\pi_2.\ \exists\pi_3.\ \varphi$, let a winning strategy be one that copies the trace π_2 into π_3. This strategy can be encoded as an Aiger circuit with 2 inputs (inputs `in_1` and `in_2` as the inputs for the first and the second universal system copies, respectively) and 1 output (output `in_3` as the input for the existential system copy). Figure 4.6 shows the corresponding strategy.

By using the strategy in the self-composition to determine the inputs to the existential system copies, these inputs are not exposed to the outside. This means that ABC cannot pick the inputs for the existential system copies; it can only decide the inputs for the universal ones.

```
aag 2 2 0 1 0
2
4
4
i0 in_1
i1 in_2
o0 in_3
c
copy in_2 into in_3
```

Figure 4.6: Strategy for a system with one input `in` and a HyperLTL formula of the form $\forall \pi_1.\ \forall \pi_2.\ \exists \pi_3.\ \varphi$ that copies trace π_2 into trace π_3.

Using this modified self-composition with the given strategy, MCHyper then builds a new circuit in the same way as in its previous version. In the resulting circuit, only the inputs of the universal system copies are exposed. The outputs are, as described above, two signals indicating a safety or a liveness violation, respectively. This circuit is then given to the backend tool ABC that tries to find an input sequence that reveals a safety or a liveness violation. If successful, this counterexample is given to the user.

Note that with our proof technique we cannot directly show that a $\forall\exists$ HyperLTL formula does not hold on some system. This is because a negative model checking result, i.e., if the model checker returns a counterexample, can either mean that the hyperproperty does not hold on the system or that we have given the wrong strategy for the existential quantifier. We can, however, prove that the negation of the formula holds to show that the original formula is violated. For this, we have to model check an $\exists\forall$ formula and provide the corresponding strategy for the existential quantifier. If we succeed in doing this, then we know that the original $\forall\exists$ formula was violated on the system, and there was no winning strategy for the existential player. The analogous argumentation applies if we want to know whether an $\exists\forall$ HyperLTL formula is violated.

We have evaluated our extension of MCHyper and submitted the corresponding artifact to the CAV artifact evaluation committee. The committee checked our artifact, its documentation and reran our experiments. Our implementation passed this evaluation so that the conference paper now uses the artifact evaluation committee seal, indicating the successful evaluation.

The incompleteness of the proof technique discussed in Chapter 3 did not cause any problems in our experiments. In fact, we did not even need to use prophecy variables as the hyperproperties we considered did not have any future dependencies. To show that verification using prophecy variables is

```
aag 1 1 0 1 0
2
2
i0 in
o0 a
c
use the input in to decide
the current value of the output a
```

Figure 4.7: System circuit that outputs arbitrary sequences of a and ¬a.

```
aag 2 2 0 1 0
2
4
2
i0 in
i1 p
o0 a
c
prophecy variable p is an additional input
```

Figure 4.8: Modified system with prophecy variable p as additional input. The system behavior is not affected by this additional input.

also possible, we start with a toy example illustrating this before we present our experimental evaluation.

4.2.1 Using Prophecy Variables

Consider again the example from Chapter 3 where the ∃-player needs to predict the next move of the ∀-player. The system we consider can produce arbitrary sequences of a and $\neg a$. An AIGER representation of this system is shown in Figure 4.7. Recall the HyperLTL formula we want to check on this system (Formula 3.1):

$$\forall \pi.\ \exists \pi'.\ \bigcirc a_\pi \leftrightarrow a_{\pi'}$$

As discussed in Chapter 3, there is no winning strategy for the ∃-player. The problem is the future dependency within the formula, i.e., that the ∃-player needs to know what the ∀-player will do in the second step. To make this information accessible to the ∃-player, we introduce a prophecy variable into the system. The prophecy variable p is added as an input to the system and does not affect the system's behavior. Figure 4.8 shows the modified system with the additional input p.

```
aag 2 2 0 2 0
2
4
4
0
i0 in_0
i1 p_0
o0 in_1
o1 p_1
c
copy value of prophecy variable p into
the input in for the existential system copy
```

Figure 4.9: Strategy using the value of the prophecy variable `p` to make the correct choice for the input `in` of the existential system copy. The input value of the prophecy variable `p` on the existential system copy is arbitrarily fixed to be constantly *false*.

The prophecy variable p can then be used by the strategy for the $\exists$-player. To ensure that the prophecy variable predicts the future correctly, we add this as an assumption to the formula we want to check, resulting in HyperLTL Formula 3.2:

$$\forall \pi.\ \exists \pi'.\ (p_\pi \leftrightarrow \bigcirc a_\pi)\ \rightarrow\ (\bigcirc a_\pi \leftrightarrow a_{\pi'}).$$

The strategy can then assume that the prophecy variable correctly predicts the value of a on π in the second step. A winning strategy then uses the value of p to set the correct value of $a_{\pi'}$. Figure 4.9 shows such a winning strategy that copies the value of p into the input of the existential system copy and, thereby, into the value of a on π'.

MCHyper successfully verifies in under a second that the HyperLTL Formula 3.2 holds on the system from Figure 4.8 using the strategy from Figure 4.9.

4.3 Experimental Evaluation

We have tested our extension of MCHyper on two different sets of examples. Our case studies use the tool to prove symmetry in mutual exclusion protocols as well as to check whether the emission control software of a car behaves as intended.

These experiments stem from previous work [17, 23]. The checked hyperproperties require a quantifier alternation when expressed in HyperLTL. Since the previous version of MCHyper could not handle quantifier alternation, the HyperLTL formulas were manually strengthened to obtain formulas in the alternation-free fragment. In this strengthening, the existential choice is fixed and used to restrict the universal quantifiers. This process is error-prone and requires deep insights into how the system works and why it satisfies the hyperproperty. The strengthened formula approximates the original formula as it is supposed to imply the original formula. This, however, needs to be proven manually as well since checking implications becomes undecidable [19].

With our extension of MCHyper, we are now able to model check the original formulas of interest directly without the need to strengthen them manually. Moreover, with our approach, there is no need to additionally prove that the strengthened formula implies the original formula of interest.

On the other hand, we need to provide an appropriate strategy as an additional input. However, the user performing the verification is familiar with the system and the hyperproperty used and most likely already has good intuition on why the hyperproperty holds on the system. Therefore, finding an appropriate strategy can be done manually. This task is not unique to our approach. In the manual strengthening of the hyperproperties in the experiments, we consider the strategy is used to restrict the universal quantifiers in the strengthened formula. So the same expertise and effort from the user are needed, whereas in our approach this manual work is nicely separated from the formula by encoding it in the strategy circuit and no additional proof of correctness for the manually strengthened formula is necessary.

4.3.1 Symmetry in Mutual Exclusion Protocols

We consider an implementation of the Bakery protocol [37], a protocol for mutual exclusion where the different processes want to access a shared critical section (e.g., a shared data structure) but only one process at a time should have access to it to avoid conflicts. The processes draw a ticket when they request to enter the critical section (*r1* means that process 1 requests access to the critical section). The process with the smallest ticket number is allowed to enter the critical section next (*g1* means that process 1 is granted access to the critical section).

Two processes may obtain the same number if they request their tickets at the same time. In this case, a mechanism for breaking the tie is needed. This mechanism might grant access to the critical section to the process with the smaller identification number (ID).

We want to check whether the protocol is symmetric. That means that when two processes swap their actions, they should also be granted access to the critical section in the opposite order. If a mutual exclusion algorithm is not symmetric, then some clients have an unfair advantage over other clients. It is, however, well known that perfect symmetry is not possible in mutual exclusion protocols [41]. This is because in the case of a tie also the symmetric execution has a tie and in both cases, the tie-breaking mechanism gives priority to the process with the smaller ID instead of picking the symmetric processes.

To achieve perfect symmetry, the tie-breaking mechanism has to be replaced by an additional input that decides, in the case of a tie, which process gets access to the critical section first. If this input gives priority to the symmetric process in the symmetric execution, perfect symmetry is achieved.

The version of the Bakery protocol for which we prove symmetry has this additional symmetry breaking input *sym_break*. Symmetry then means that for every protocol execution, the symmetric execution is also possible in the protocol. This is a hyperproperty as it relates the two symmetric executions. In HyperLTL, symmetry for the mutual exclusion protocol can be formalized as:

$$\begin{aligned}\forall \pi.\ \exists \pi'.\ \square((\mathit{r1}_\pi \leftrightarrow \mathit{r2}_{\pi'}) \land (\mathit{r2}_\pi \leftrightarrow \mathit{r1}_{\pi'})\\ \land (\mathit{g1}_\pi \leftrightarrow \mathit{g2}_{\pi'}) \land (\mathit{g2}_\pi \leftrightarrow \mathit{g1}_{\pi'})).\end{aligned} \tag{4.1}$$

The manually strengthened version of this property used in the previous case study [23] was:

$$\begin{aligned}\forall \pi.\ \forall \pi'.\ \square((\mathit{r1}_\pi \leftrightarrow \mathit{r2}_{\pi'}) \land (\mathit{r2}_\pi \leftrightarrow \mathit{r1}_{\pi'})\\ \land (\mathit{sym\_break}_\pi \not\leftrightarrow \mathit{sym\_break}_{\pi'}))\\ \rightarrow \square((\mathit{g1}_\pi \leftrightarrow \mathit{g2}_{\pi'}) \land (\mathit{g2}_\pi \leftrightarrow \mathit{g1}_{\pi'})).\end{aligned}$$

This version states that all pairs of executions with symmetric requests to enter the critical section should grant access to this section symmetrically if also the symmetry breaking input is symmetrical. Even though this formula is universally quantified, it does not consider all arbitrary pairs of traces but only those where the requests are symmetric. In the previous case study [23] it was already shown that this hyperproperty holds for the Bakery protocol with the additional symmetry breaking input. This result implies that the original symmetry requirement holds for the Bakery protocol. This implication, however, has to be shown in an additional proof. Otherwise, it is not guaranteed that the proof of the manually strengthened formula allows any conclusions about the original formula.

```
aag 3 3 0 3 0
2
4
6
4
2
7
i0 r1_0
i1 r2_0
i2 sym_break_0
o0 r1_1
o1 r2_1
o2 sym_break_1
c
give symmetric requests and symmetry breaking input
```

Figure 4.10: Strategy observing the requests on the universally quantified trace and setting the inputs on the existentially quantified trace to the symmetric values.

From this strengthened formula, we can extract the strategy. In fact, the premise that restricts the accepted choices for the trace π' represents the strategy. The strategy here is always to make sure that the requests to enter the critical section are given symmetrically and the protocol execution makes sure that the grants are given out symmetrically as well. The strategy circuit shown in Figure 4.10 swaps the requests of the two processes and picks the opposite value for the symmetry breaking input on the existentially quantified trace.

In addition to finding the strategy and encoding it in the manually strengthened formula, with the old version of MCHyper, it is also necessary to prove the correctness of the intended implication. With the extended version of MCHyper we now can model check the symmetry formula with the quantifier alternation directly. The symmetry breaking input *sym_break* does not even occur in the formula explicitly. The additional input to MCHyper is the strategy we extracted from the manually strengthened formula shown in Figure 4.10.

We have checked symmetry in the Bakery protocol with the extended version of MCHyper. Table 4.1 shows our verification results. We have checked a smaller variant of the Bakery protocol supporting three clients and a bigger version supporting five clients. Naturally, the bigger version of the protocol needs more latches to represent the larger state space. On these systems,

#Processes	#Latches	Hyperproperty	Time [s]
3	47	Symmetry	50.6
		Symmetry’	27.5
5	90	Symmetry	461.3
		Symmetry’	472.3

Table 4.1: Experimental results for MCHyper on the mutual exclusion benchmarks. All experiments used the IC3 option for abc.

we have checked the HyperLTL Formula 4.1 (Symmetry) and a version of symmetry using the *weak until* operator (Symmetry’):

$$\begin{aligned}\forall \pi.\ \exists \pi'.\ &((g1_\pi \leftrightarrow g2_{\pi'}) \wedge (g2_\pi \leftrightarrow g1_{\pi'}))\\ &\mathcal{W}\ \ ((r1_\pi \not\leftrightarrow r2_{\pi'}) \vee (r2_\pi \not\leftrightarrow r1_{\pi'})).\end{aligned}$$

This formula states that the grants have to be given symmetrically unless the requests were not made symmetrically at some point. In the case where the requests are always symmetric, this formula is equivalent to Formula 4.1.

To prove these hyperproperties, we have extracted the corresponding strategies from the manually strengthened versions. The used strategies ensure that the requests are made symmetrically and that the symmetry breaking input is symmetric on both traces, similar to the strategy shown in Figure 4.10. For both hyperproperties, Symmetry and Symmetry’, on one version of the Bakery protocol, the same strategy can be used as the system’s inputs, and outputs are the same.

We were able to show within a few minutes that the Bakery protocol with the symmetry breaking input to break ties is indeed symmetric. The times needed for the verification of the Bakery protocol are comparable to the ones obtained in the previous case study [23]. This is not surprising since we are solving the same problem and only move the strategy from the strengthened formula into a dedicated strategy input. However, we now have directly checked the formula of interest and do not need any additional proofs to guarantee correctness.

4.3.2 Software Doping

The second set of experiments [17] is inspired by the emission scandal in the automotive industry from 2015. This scandal coined the term “software doping” as it revealed that the diesel emission regulations were violated on purpose by some car manufacturers. In their cars, the exhaust emission control module in the electronic control unit of the car is “doped”, i.e., it behaves differently when being tested and when being driven on the road. In the test scenario, the car meets the diesel emission regulations; thus, it passes the test. When being driven on the road, however, the car emits much more exhaust gasses. The exhaust emission control module can regulate the emission of the nitrogen oxides (NOx) by injecting diesel exhaust fluid (DEF) into the exhaust pipeline that reacts with the nitrogen oxides. This reaction is called selective catalytic reduction (SCR), and it uses the DEF, an aqueous urea solution, to initiate a chemical reaction with the nitrogen oxides that results in water and harmless nitrogen as end products that can then be released into the environment.

Intuitively, when checking whether a system is doped, we are looking for two different sequences of inputs that are ‘close’ but lead to two very different sequences of outputs. In clean software, we expect that similar inputs yield to similar outputs even though one input sequence might be a test scenario while another input sequence corresponds to driving on the road.

To decide whether two sequences are similar (or close), we need to have a notion of distance. We capture this by a distance function $\hat{d}$ that takes two values and computes their distance. Different distance functions might be appropriate, depending on the scenario. The distance function can be applied to the inputs and the outputs or two different distance functions can be used. In our scenario, we use the same distance function for inputs and outputs and, given two values, the function returns their difference. This distance can then be compared to a threshold κ, for example, $\hat{d}(o, o') \leq \kappa$.

In this set of experiments, we consider a clean and a doped version of an exhaust emission control module. We model the modules as reactive systems that constantly read the throttle value and calculate the amount of DEF that is added to the exhaust pipeline. This determines the amount of nitrogen oxide that will be emitted, although this process is not perfect. To account for this variability, we use non-determinism in the system to model the final amount of nitrogen oxide that is emitted. A more detailed description of the systems can be found in the previous case study [17].

The hyperproperty we want to check is called robust cleanness. It captures the intuition described above by requiring that the distance between the outputs always is within some threshold κ_o unless at some point the distance in the inputs is bigger than some threshold κ_i on the allowed input distance. The following HyperLTL formula formalizes robust cleanness:

$$\begin{aligned}\forall \pi.\ \forall \pi'.\ \exists \pi''.\ & \square \big(i_{\pi'} = i_{\pi''}\big) \\ & \wedge\ \big(\hat{d}(o_\pi, o_{\pi''}) \leq \kappa_o\ \ \mathcal{W}\ \ \hat{d}(i_\pi, i_{\pi''}) > \kappa_i\big). \end{aligned} \tag{4.2}$$

```
aag 0 0 0 2 0
0
1
o0 in1
o1 in2
c
always set in1 to false and in2 to true
```

Figure 4.11: Strategy that encodes a constant trace where inputs `in1` and `in2` are always set to *false* and *true*, respectively.

We would like to compare the inputs and outputs of the two universally quantified traces. Due to the non-determinism in the models, we cannot guarantee that every pair of traces satisfies our hyperproperty. To account for that, we ask for a third, existentially quantified trace that has to agree with π' on the inputs and for which our hyperproperty holds.

Since the distance function is not necessarily symmetric, we also need to consider the symmetric formula where the existentially quantified trace agrees with π on the inputs:

$$\begin{aligned} &\forall \pi.\ \forall \pi'.\ \exists \pi''.\ \square\big(i_\pi = i_{\pi''}\big) \\ &\qquad\qquad \wedge\ \big(\hat{d}(o_{\pi''}, o_{\pi'}) \leq \kappa_o\ \mathcal{W}\ \hat{d}(i_{\pi''}, i_{\pi'}) > \kappa_i\big). \end{aligned} \tag{4.3}$$

When we refer to robust cleanness in the following, we mean the conjunction of Formulas 4.2 and 4.3.

This case study has been explored before [17], and the previous version of MCHyper has been used to automatically check whether the emission control software is robustly clean. Again, because the previous version of MCHyper could not handle quantifier alternation, the hyperproperty had to be manually strengthened into the alternation-free fragment. To prove that the clean exhaust emission control module is indeed robustly clean the following formula was checked:

$$\forall \pi.\ \forall \pi'.\ \big(\hat{d}(o_\pi, o_{\pi'}) \leq \kappa_o\ \mathcal{W}\ \hat{d}(i_\pi, i_{\pi'}) > \kappa_i\big).$$

This is a stronger hyperproperty as it does not allow for violations of the specification that result only from non-determinism in the system.

To prove that the doped exhaust emission control module violates robust cleanness, we have to show that it satisfies the negation of this hyperproperty as we have explained in Section 4.2. The negations of Formulas 4.2 and 4.3 both are of the form $\exists \pi.\ \exists \pi'.\ \forall \pi''.\ \varphi$. They are strengthened into universal formulas by adding a premise ψ that assumes the input values for the first two quantified traces are set so some determined values. The strategy picking

System	Precision	#Latches	Hyperproperty	Time [s]
clean	medium	17	Robust Cleanness	1.8
	high	23		53.4
doped	medium	19	¬ Robust Cleanness	2.8
	high	25		160.1

Table 4.2: Experimental results for MCHyper on the software doping benchmarks. All experiments used the IC3 option for ABC.

the witnesses for the existential traces is again encoded into the formula. Additionally, it has to be proven that these two fixed traces indeed exist in our system. Thus, a third alternation-free formula of the form $\exists\pi.\ \exists\pi'.\ \psi$ has to be checked to show that the strengthening is correct.

Using the extended version of MCHyper, we can now directly model check robust cleanness and its negation on the two systems. To do so, we extract the strategies that were encoded in the manual strengthening of the formulas.

We model check two different versions of the exhaust emission control modules. The difference is the precision with which they capture the throttle input and the NOx output. Higher precision in the system means that more bits are needed to represent a single value and, therefore, the state space, i.e., the number of latches increases.

Consider the clean version of the exhaust emission control module first. The strategy we extracted from the manual strengthening of the robust cleanness requirement copies one of the universally quantified traces into the existentially quantified trace. Figure 4.6 shows a strategy that does this for a system with only one input.

For the doped version of the exhaust emission control module, the negation of robust cleanness has to be checked. Here, the strategies extracted from the manual strengthening encode a fixed traces for the two existential quantifiers. Figure 4.11 shows an example of a strategy that encodes a fixed trace.

Table 4.2 displays our verification results. Also for this case study, the times needed to verify the systems with the extended version of MCHyper are comparable to the times needed to verify the systems with the previous version against the manually strengthened formulas. However, we again gain the additional correctness guarantee without the need to verify that our strengthening was correct as it had to be done previously.

Acknowledgements. We thank Sebastian Biewer for providing the data from the previous software doping case study.

4.4 Tutorial and Online Interface

The extended version of MCHyper is available via its online interface[1]. On this website, everyone can try the tool in their browser. We provide some examples as well as an editor that allows the user to submit their inputs to the tool. The instance is then solved on our servers at the Reactive Systems Group at Saarland University, and the verification results are reported back to the user.

We also provide a detailed tutorial explaining how to use MCHyper. In this tutorial, we explain the online tool interface, the different input formats for MCHyper, and how the tool works. The tutorial also covers the extension of MCHyper to formulas with quantifier alternation.

Acknowledgements. The online tool interface was developed, under our supervision, by our student assistants Jens Kreber (backend development) and Benedict Strube (frontend development and tutorial).

[1]https://www.react.uni-saarland.de/tools/online/MCHyper/

Chapter 5

Conclusions

In this thesis, we have explored the model checking problem of HyperLTL. In particular, we have presented a practical model checking technique for HyperLTL formulas with one quantifier alternation. These formulas specify hyperliveness properties that previously could not be model checked automatically. The model checking tool MCHyper was restricted to the alternation-free fragment of HyperLTL. This fragment cannot express hyperliveness properties that require a quantifier alternation in the HyperLTL formula. Hyperproperties like generalized noninterference, symmetry or robust cleanness, however, fall into this fragment.

We have extended the tool MCHyper to be able to handle formulas with up to one quantifier alternation. This extended version of MCHyper is the first practical method for the verification of these HyperLTL formulas. The main idea behind our extension is a shift in perspective to a *game-theoretic view.* We use a strategy to determine the choices of the ∃-player, thereby substituting strategic choice for existential choice. The strategy for the ∃-player observes the choices for the preceding universally quantified traces and, depending on these, chooses the moves for the ∃-player.

If an appropriate strategy exists with which the verification succeeds, we can conclude that the hyperproperty holds on the system. This approach, however, is not complete. That means that there are systems that satisfy a hyperproperty but for which no appropriate strategy exists. This might happen if the correct existential choices depend on the future behavior of the ∀-player. We have explored this issue in more detail by discussing how prophecy variables can be used to resolve future dependencies.

Prophecy variables can be used to resolve these dependencies. Nevertheless, using prophecy variables still does not suffice to obtain a complete proof method. The core problem leading to the incompleteness result for our model checking approach for HyperLTL is that HyperLTL, just like LTL, cannot express counting properties. Thus, hyperproperties that require to distinguish, for example, between even and odd positions cannot be proven using our technique.

We have studied an extension of HyperLTL that can express counting properties and for which we obtained a completeness result for our model checking approach.

Despite this incompleteness result for HyperLTL, the presented model checking technique is useful in practice. We have shown this in our experimental evaluation where we have checked symmetry in mutual exclusion algorithms and robust cleanness for emission control software in cars. In our experiments, the incompleteness was never a problem. We did not even have to use prophecy variables since the hyperproperties do not have any future dependencies.

The key advantage of our approach is that it avoids the complementation of the Büchi automaton that is built in the automata-based model checking algorithm for HyperLTL. Complementing a Büchi automaton is exponential and, in the automata-based algorithm, it has to be done once for every quantifier alternation in the formula. With our approach, the user instead has to provide an appropriate strategy for the existential player. Providing a sufficiently small strategy leads to a significant gain in performance.

It is also possible to automatically synthesize a winning strategy [15] for a given system and hyperproperty. Instead of giving the system, this can also be synthesized together with the strategy. In both cases, the separation of system, strategy and HyperLTL formula is exploited.

In theory, our model checking technique using strategic choice can be extended to an arbitrary number of quantifier alternations. This has, however, not yet been implemented and most hyperproperties of interest used in the literature only require at most one quantifier alternation. Searching for practically relevant hyperproperties with more than one quantifier alternation and extending the implementation of MCHyper even further to handle these hyperproperties is left as interesting future work.

Moreover, it would be interesting to learn about real-world problems where future dependencies occur or where the incompleteness of our approach is an issue. Studying the lack of completeness of our proof technique, as well as the completeness for the extended version of HyperLTL, in more detail is also left as future work.

Bibliography

[1] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991. doi:10.1016/0304-3975(91)90224-P.

[2] Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider, editors. *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985, Munich, Germany*, volume 190 of *Lecture Notes in Computer Science*, 1985. Springer. ISBN 3-540-15216-4. doi:10.1007/3-540-15216-4.

[3] Bowen Alpern and Fred B. Schneider. Defining Liveness. Technical report, Ithaca, NY, USA, 1984.

[4] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, Sep 1987. ISSN 1432-0452. doi:10.1007/BF01782772.

[5] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure Information Flow by Self-Composition. In *Proceedings of CSFW*, pages 100–114. IEEE Computer Society, 2004.

[6] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification. In *Proceedings of LFCS*, volume 7734 of *LNCS*, pages 29–43. Springer, 2013. doi:10.1007/978-3-642-35722-0_3.

[7] Armin Biere. Specification of the AIGER Format. `http://fmv.jku.at/aiger/FORMAT`. Online; accessed: 2019-09-18.

[8] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness Checking as Safety Checking. *Electr. Notes Theor. Comput. Sci.*, 66(2):160–177, 2002. doi:10.1016/S1571-0661(04)80410-9.

[9] Robert K. Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *Proceedings of CAV*, volume

6174 of *LNCS*, pages 24–40. Springer, 2010. doi:10.1007/978-3-642-14295-6_5.

[10] Julius Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. *Internat. Congress on Logic, Methodology and Philosophy of Science*, 1960.

[11] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1982. Springer-Verlag. ISBN 3-540-11212-X. URL `http://dl.acm.org/citation.cfm?id=648063.747438`.

[12] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

[13] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal Logics for Hyperproperties. In *Proceedings of POST*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.

[14] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. The Hierarchy of Hyperlogics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. ISBN 978-1-7281-3608-0. doi:10.1109/LICS.2019.8785713.

[15] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying Hyperliveness. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 121–139, Cham, 2019. Springer International Publishing. ISBN 978-3-030-25540-4. doi:10.1007/978-3-030-25540-4_7.

[16] Byron Cook, Heidy Khlaaf, and Nir Piterman. On Automation of CTL* Verification for Infinite-State Systems. In *Proceedings of CAV*, volume 9206 of *LNCS*, pages 13–29. Springer, 2015. doi:10.1007/978-3-319-21690-4_2.

[17] Pedro R. D'Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. Is Your Software on Dope? - Formal Analysis of Surreptitiously "enhanced" Programs. In *Proceedings of ESOP*, volume 10201 of *LNCS*, pages 83–110. Springer, 2017. doi:10.1007/978-3-662-54434-1_4.

[18] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time Temporal Logic. *J. ACM*, 33(1):151–178, January 1986. ISSN 0004-5411. doi:10.1145/4904.4999.

[19] Bernd Finkbeiner and Christopher Hahn. Deciding Hyperproperties. In *Proceedings of CONCUR*, volume 59 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CONCUR.2016.13.

[20] Bernd Finkbeiner and Markus N. Rabe. The linear-hyper-branching spectrum of temporal logics. *it - Information Technology*, 56:273–279, November 2014.

[21] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6): 519–539, 2013. doi:10.1007/s10009-012-0228-z.

[22] Bernd Finkbeiner and Martin Zimmermann. The First-Order Logic of Hyperproperties. In Heribert Vollmer and Brigitte Vallée, editors, *34th Symposium on Theoretical Aspects of Computer Science (STACS 2017)*, volume 66 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-028-6. URL `http://drops.dagstuhl.de/opus/volltexte/2017/7003`.

[23] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for Model Checking HyperLTL and HyperCTL*. In *Proceedings of CAV*, volume 9206 of *LNCS*, pages 30–48. Springer, 2015. doi:10.1007/978-3-319-21690-4_3.

[24] Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. EAHyper: Satisfiability, Implication, and Equivalence Checking of Hyperproperties. In *Proceedings of CAV*, volume 10427 of *LNCS*, pages 564–570. Springer, 2017. doi:10.1007/978-3-319-63390-9_29.

[25] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring Hyperproperties. In *Proceedings of RV*, volume 10548 of *LNCS*, pages 190–207. Springer, 2017. doi:10.1007/978-3-319-67531-2_12.

[26] Bernd Finkbeiner, Christopher Hahn, and Tobias Hans. MGHyper: Checking Satisfiability of HyperLTL Formulas Beyond the $\exists^*\forall^*$ Fragment. In *Proceedings of ATVA*, volume 11138 of *LNCS*, pages 521–527. Springer, 2018. doi:10.1007/978-3-030-01090-4_31.

[27] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. Synthesizing Reactive Systems from Hyperproperties. In *Proceedings of CAV*, volume 10981 of *LNCS*, pages 289–306. Springer, 2018. doi:10.1007/978-3-319-96145-3_16.

[28] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. RVHyper: A Runtime Verification Tool for Temporal Hy-

perproperties. In *Proceedings of TACAS*, volume 10806 of *LNCS*, pages 194–200. Springer, 2018. doi:10.1007/978-3-319-89963-3_11.

[29] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. Model Checking Quantitative Hyperproperties. In *Proceedings of CAV*, volume 10981 of *LNCS*, pages 144–163. Springer, 2018. doi:10.1007/978-3-319-96145-3_8.

[30] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of S&P*, pages 11–20. IEEE Computer Society, 1982. doi:10.1109/SP.1982.10014.

[31] Christopher Hahn, Marvin Stenger, and Leander Tentrup. Constraint-Based Monitoring of Hyperproperties. In *Proceedings of TACAS*, volume 11428 of *LNCS*, pages 115–131. Springer, 2019. doi:10.1007/978-3-030-17465-1_7.

[32] Marieke Huisman, Pratik Worah, and Kim Sunesen. A Temporal Logic Characterisation of Observational Determinism. In *Proceedings of CSFW*, page 3. IEEE Computer Society, 2006. doi:10.1109/CSFW.2006.6.

[33] Abdessamad Jarrar and Youssef Balouki. Formal modeling of a complex adaptive air traffic control system. *Complex Adaptive Systems Modeling*, 6(1):6, Sep 2018. ISSN 2194-3206. doi:10.1186/s40294-018-0056-4.

[34] Felix Klein and Martin Zimmermann. How Much Lookahead is Needed to Win Infinite Games? In *Proceedings of ICALP*, volume 9135 of *LNCS*, pages 452–463. Springer, 2015. doi:10.1007/978-3-662-47666-6_36.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[36] Orna Kupferman and Moshe Y. Vardi. Weak Alternating Automata Are Not That Weak. *ACM Trans. Comput. Logic*, 2(3):408–429, July 2001. ISSN 1529-3785. doi:10.1145/377978.377993.

[37] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, 1974. doi:10.1145/361082.361093.

[38] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. doi:10.1109/TSE.1977.229904.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[40] Nancy A. Lynch and Frits W. Vaandrager. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.*, 121(2):214–233, 1995. doi:10.1006/inco.1995.1134.

[41] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems - Safety.* Springer, 1995. ISBN 978-0-387-94459-3.

[42] Daryl McCullough. Noninterference and the Composability of Security Properties. In *Proceedings of S&P*, pages 177–186. IEEE Computer Society, 1988. doi:10.1109/SECPRI.1988.8110.

[43] Robert McNaughton and Seymour A. Papert. *Counter-Free Automata (M.I.T. Research Monograph No. 65).* The MIT Press, 1971. ISBN 0262130769.

[44] Satoru Miyano and Takeshi Hayashi. Alternating Finite Automata on ω-Words. *Theor. Comput. Sci.*, 32:321–330, 1984. doi:10.1016/0304-3975(84)90049-5.

[45] D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *[1988] Proceedings. Third Annual Symposium on Logic in Computer Science*, pages 422–427, July 1988. doi:10.1109/LICS.1988.5139.

[46] Andew Myers. Meltdown, Spectre, and why hardware can be correct yet insecure. `https://andrumyers.wordpress.com/2018/01/17/meltdown-spectre-and-how-hardware-can-be-correct-but-insecure/`. Online; accessed: 2019-09-09.

[47] John von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, Dec 1928. ISSN 1432-1807. doi:10.1007/BF01448847.

[48] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. doi:10.1109/SFCS.1977.32.

[49] Markus N. Rabe. *A Temporal Logic Approach to Information-Flow Control.* PhD thesis, Saarland University, 2016. URL `http://scidok.sulb.uni-saarland.de/volltexte/2016/6387/`.

[50] Computing Research and Education Association of Australasia. CORE18 Ranking for CAV'19. http://portal.core.edu.au/conf-ranks/?search=CAV&by=all&source=CORE2018&sort=atitle&page=1. Online; accessed: 2019-09-09.

[51] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *J. ACM*, 32(3):733–749, July 1985. ISSN 0004-5411. doi:10.1145/3828.3837.

[52] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2):217 – 237, 1987. ISSN 0304-3975. URL http://www.sciencedirect.com/science/article/pii/0304397587900089.

[53] Aravinda Prasad Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems.* PhD thesis, Cambridge, MA, USA, 1983. AAI8403047.

[54] Thoralf Skolem. Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen. Videnskapsselskapet Skrifter, I. Matematisk-naturvidenskabelig Klasse 4 (1920).

[55] Tachio Terauchi and Alexander Aiken. Secure Information Flow as a Safety Problem. In *Proceedings of SAS*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005. doi:10.1007/11547662_24.

[56] Ron van der Meyden and Chenyi Zhang. Algorithmic Verification of Noninterference Properties. *Electr. Notes Theor. Comput. Sci.*, 168: 61–75, 2007. doi:10.1016/j.entcs.2006.11.002.

[57] Moshe Y. Vardi. *Alternating Automata and Program Verification*, pages 471–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. ISBN 978-3-540-49435-5. doi:10.1007/BFb0015261.

[58] Clifford Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/. Online; accessed: 2019-09-18.

[59] Yang Zhao and Kristin Yvonne Rozier. Formal specification and verification of a coordination protocol for an automated air traffic control system. *Science of Computer Programming*, 96:337 – 353, 2014. ISSN 0167-6423. URL http://www.sciencedirect.com/science/article/pii/S016764231400166X. Special Issue on Automated Verification of Critical Systems (AVoCS 2012).

Appendix A

Completeness Proof

We present the full proof of Theorem 3.4.

Proof Let a finite-state transition system $\mathcal{S}$ and a $\forall^n\exists^m$ safety HyperLTL formula $\psi \coloneqq \forall\pi_1 \dots \forall\pi_n.\ \exists\pi'_1 \dots \exists\pi'_m.\ \varphi$ be given where φ is a safety LTL formula. Assume that $\mathcal{S} \vDash \psi$. We show that there always exists a winning strategy using prophecy variables with which the model checking succeeds for the modified formula containing the prophecy variables. For that, we first show how to obtain a finite set of prophecy variables that capture the necessary information about the future. Then, we show how to include this information into the formula that we want to model check. Lastly, we describe a winning strategy that uses the captured information about the future to make appropriate choices in each step.

Find prophecy variables. Build the non-deterministic safety automaton [4] $\mathcal{A}_\varphi = \langle Q, q_0, \delta, B\rangle$ for the quantifier-free HyperLTL formula φ. This automaton over $\Sigma = (2^{I\cup O})^{n+m}$ takes tuples of trace positions as input.

Next, build the product construction of the self-composed system $\mathcal{S}^{n+m}$ and the automaton $\mathcal{A}_\varphi$ yielding a non-deterministic safety automaton $\mathcal{S}^{n+m} \times \mathcal{A}_\varphi$ defined as $\langle Q^\times, q_0^\times, \delta^\times, B^\times\rangle$ over $\Sigma^\times = \Upsilon^{n+m}$ where $Q^\times = T^{n+m} \times Q$ is the finite set of states, $q_0^\times = (\vec{t_0}, q_0)$ is the initial state, $(\vec{t'}, q') \in \delta^\times((\vec{t}, q), \vec{v})$ for all $\vec{t'} = \tau_{n+m}(\vec{t}, \vec{v})$ and $q' \in \delta(q, (l \circ \vec{t}) \uplus \vec{v})$ is the transition relation and $B^\times = \{(\vec{t}, q) | q \in B\}$ is the set of accepting states.

Apply the existential projection to the last m positions of the inputs of $\mathcal{S}^{n+m} \times \mathcal{A}_\varphi$ resulting in a non-deterministic safety automaton over $\Sigma_\exists = \Upsilon^n$ $\mathcal{A}_\exists = \langle Q^\times, q_0^\times, \delta_\exists, B^\times\rangle$, where $\delta_\exists$ is the transition relation of this automaton with $(\vec{t'}, q') \in \delta_\exists((\vec{t}, q), \vec{v})$ if there exists a $\vec{u} \in \Upsilon^m$ such that $\vec{t'} = \tau_{n+m}(\vec{t}, \vec{v} \cdot \vec{u})$ and $q' \in \delta(q, (l \circ \vec{t}) \uplus (\vec{v} \cdot \vec{u}))$. This automaton accepts all input sequences determining the n universally quantified traces for which there are adequate choices for the existentially quantified traces such that the formula is satisfied. Since we assume that the formula holds the language is actually equal to $(\Upsilon^n)^\omega$. For every state $q \in Q_\exists$ we calculate the language of the automaton $\mathcal{A}_\exists^q$ obtained by considering state q as the initial state of $\mathcal{A}_\exists$. The languages of all these automata $\mathcal{A}_\exists^q$ are again subsets of $(\Upsilon^n)^\omega$ and capture the set of future universal choices for which the existential quantifiers can make appropriate choices to ensure the satisfaction of the formula when starting in q.

We add one prophecy variable p^q for every state $q \in Q_\exists$ as auxiliary variables to our system and let p^q guess whether the suffix of the universal traces is in $\mathcal{L}(\mathcal{A}_\exists^q)$. When p^q guesses correctly and it is *true* for some state q, this guarantees that, given that all traces together reached state q, there is an accepting run on $\mathcal{A}_\exists^q$ for the future behavior of the universally quantified traces. This, in turn, means that there is an adequate choice to extend the existentially quantified traces such that they together with the universally quantified traces satisfy φ. Note that, as required, no assumption about the future behavior of the existentially quantified traces is made.

Modify formula. To include the prophecy variables into the HyperLTL formula, we construct formulas $\exists at_{q_0} \dots \exists at_{q_x}.\ \varphi_q$ that have the same language as $\mathcal{A}_\exists^q$ where we quantify one fresh atomic proposition for every of the $x+1$ states in $Q^\times$. The formula $\exists at_{q_0} \dots \exists at_{q_x}.\ \varphi_q$ intuitively encodes all accepting runs in the automaton $\mathcal{A}_\exists^q$ and formally is defined as follows:

$$\exists at_{q_0} \dots \exists at_{q_x}.\ at_q \tag{A.1}$$

$$\land\ \square\,\Big(\bigvee_{(s,\vec{v},s')\in\delta_\exists} at_s \land \bigcirc\ at_{s'} \land \big(\bigwedge_{i=1}^{n} \vec{v}[k]_{\pi_k}\big)\Big) \tag{A.2}$$

$$\land\ \square\,\Big(\bigwedge_{s\in Q^\times}\ \bigwedge_{s'\in Q^\times\setminus\{s\}} \neg(at_s \land at_{s'})\Big) \tag{A.3}$$

$$\land\ \square\,\Diamond \bigvee_{s\in B^\times} at_s \tag{A.4}$$

(A.1) ensures that every run starts in the current initial state, (A.2) ensures that the transition relation of $\mathcal{A}_\exists^q$ is followed and that the inputs correspond to the n universally quantified traces where π_k is the k-th universally quantified trace, (A.3) ensures that the run is always at exactly one state and (A.4) ensures that the run is accepting, i.e., that infinitely many accepting states are visited.

We want every prophecy variable p^q to be equivalent to the generated formula $\exists at_{q_0} \dots \exists at_{q_x}.\ \varphi_q$. We include these equivalences as the premise into the formula as described resulting in the following formula:

$$\forall\pi_1 \dots \forall\pi_n.\ \exists\pi'_1 \dots \exists\pi'_m.\ \big(\square \bigwedge_{q\in Q^\times} (p^q_{\pi_1} \leftrightarrow \exists at_{q_0} \dots \exists at_{q_x}.\ \varphi_q)\big) \rightarrow \varphi.$$

This formula explicitly captures all necessary information about the future behavior of the universally quantified traces by using the QPTL-like subformulas.

Construct winning strategy. We describe a strategy that uses the information captured by the prophecy variables to decide the next inputs $\vec{v_\exists} \in \Upsilon^m$ for the existentially quantified traces given the current inputs $\vec{v_\forall} \in \Upsilon^n$ for the universally quantified traces. The strategy assumes that the prophecy variables correctly predict the future. To obtain a deterministic strategy, fix some order on Υ^m and $Q^\times$. Our strategy $\sigma : (\Upsilon^n)^* \to \Upsilon^m$ is defined as $\sigma(\vec{\varepsilon_\forall}) = \vec{\varepsilon_\exists}$ and $\sigma(\vec{u_\forall}\vec{v_\forall}) = \vec{v_\exists}$ where $\vec{v_\exists}$ is the first input in Υ^m that can make a transition from the current state $q = \sigma'(\vec{u_\forall})$ to the goal state $q' = \sigma'(\vec{u_\forall}\vec{v_\forall})$, i.e., $(q, \vec{v_\forall} \cdot \vec{v_\exists}, q') \in \delta^\times$. If no such input $\vec{v_\exists}$ exists, set $\sigma(\vec{u_\forall}\vec{v_\forall}) = \vec{\varepsilon_\exists}$. The strategy σ uses the function $\sigma' : (\Upsilon^n)^* \to Q^\times$ that updates the state in the safety automaton $\mathcal{S}^{n+m} \times \mathcal{A}_\varphi$. This function is defined as $\sigma'(\vec{\varepsilon_\forall}) = q_0^\times$ and $\sigma'(\vec{u_\forall}\vec{v_\forall}) = q''$ where we obtain q'' as follows: Let $q = \sigma'(\vec{u_\forall})$ be the current system state. In the given order, try all inputs $\vec{v_\exists}$ and check the set of possible successor states $\delta^\times(q, \vec{v_\forall} \cdot \vec{v_\exists})$. In the given order, check for all states q' in that set whether $p^{q'}$ is *true* (i.e., whether $p^{q'} \in l(q)[1]$). Pick the first state q' for which this holds and let $q'' = q'$. If no such state exists, set $q'' = q_0^\times$.

We prove by contradiction that the strategy σ is winning. Assume the strategy is not winning. That means, there is a sequence of universal inputs $\vec{w_\forall} \in (\Upsilon^n)^\omega$ for which the strategy σ yields, step by step, a sequence of existential inputs $\vec{w_\exists} \in (\Upsilon^m)^\omega$ such that the resulting trace ρ, obtained from $\vec{w}$ where $\vec{w}[i] = \vec{w_\forall}[i] \cdot \vec{w_\exists}[i]$ for all positions i, through the self-composed system $\mathcal{S}^{n+m}$ does not satisfy the following formula:

$$\forall \pi^*.\ zip((\square \bigwedge_{q \in Q^\times} (p^q_{\pi_1} \leftrightarrow \exists at_{q_0} \ldots \exists at_{q_x}.\ \varphi_q)) \to \varphi, \pi_1, \ldots, \pi_n, \pi'_1, \ldots, \pi'_m).$$

■

This means the values of the prophecy variables always predict the future correctly, but there is no accepting run on ρ through the automaton $\mathcal{A}_\varphi$. We show, however, that it is possible to construct such an accepting run for the given trace, thus contradicting the assumption that the formula was violated. In fact, the accepting run we are looking for is the one calculated on the fly by the function σ' which keeps track of the state in $\mathcal{S}^{n+m} \times MH(\mathcal{A}_\varphi)$. The second component of these states is a state of $\mathcal{A}_\varphi$. Since whether a run is accepting in $\mathcal{S}^{n+m} \times \mathcal{A}_\varphi$ always depends on whether the run in the $\mathcal{A}_\varphi$ component is accepting, we will, in the following, reason about runs in $\mathcal{S}^{n+m} \times \mathcal{A}_\varphi$. The sequence of states generated by σ' is a valid run through $\mathcal{S}^{n+m} \times \mathcal{A}_\varphi$ by definition of σ', and it is accepting. To see that it is an accepting run and thus, our strategy is winning, recall that $\mathcal{S}^{n+m} \times \mathcal{A}_\varphi$ is the self-composition of the system combined with the safety automaton for the quantifier-free HyperLTL formula φ that is a safety property. Given the universally quantified traces, we show by *induction over the length l of the prefix* that σ extends the existentially quantified traces in a way that the run constructed by σ' never leaves the set of safe states. Thus, there never is a bad prefix, so the run is accepting and the safety property φ holds for these traces.

In the *base case*, we consider the prefix of $\vec{w_\forall}$ with length $l = 1$, i.e., we have some $\vec{v} = \vec{q_\forall}[0] \in \Upsilon^n$. Calculate $\sigma(\vec{v}) = \sigma(\vec{\varepsilon_\forall} \cdot \vec{v})$. For that, $\sigma'(\vec{\varepsilon_\forall}) = q_0^\times$ is computed, as well as $\sigma'(\vec{v})$. During the computation of the latter, inputs $\vec{v'} \in \Upsilon^m$ to extend the existentially quantified traces and a corresponding successor state $q' \in Q^\times$ for which $p^{q'}$ holds are found. We know that these $\vec{v'}$ and q' exist so that the default option of σ' is not used, because we assume that the hyperproperty holds and the prophecy variables always guess correctly. Knowing that the hyperproperty holds means that from the initial state there is, for all possible universally quantified traces, a corresponding choice of the existentially quantified traces that together generate an accepting run through $\mathcal{S}^{n+m} \times \mathcal{A}_\varphi$. Because such an accepting run must exist, there has to be one reachable next state that lies on this run. For this successor state the corresponding prophecy variable has to be *true* as we know that this state lies on an accepting run and the prophecy variables always guess correctly. Thus, at least this one prophecy variable has to be *true* (if there is just one accepting run) and σ' uses the found successor state for q' and the existential inputs that lead to this state for $\vec{v'}$. This $\vec{v'}$ is also the value that $\sigma(\vec{v})$ returns for the first position of the existentially quantified traces. We know that from the initial state in the non-deterministic safety automaton $\mathcal{S}^{n+m} \times MH(\mathcal{A}_\varphi)$ we can reach state q' with $\vec{v} \cdot \vec{v'}$ (σ' checked this). Moreover, we know that q' is part of an accepting run that has to exist because $p^{q'}$ is *true*. Thus, $\vec{v} \cdot \vec{v'}$ cannot be a bad prefix for the property φ and we know that choosing $\vec{v'}$ does not violate the property.

The *induction hypothesis* now states that for prefixes of all lengths l the choices of σ create a run that is safe up to the current state and the choices did not lead to a bad prefix yet. Moreover, we know that the prophecy variable from the previous step promised that there exists an accepting run from the current state for the predicted future behavior of the universally quantified traces. In the *induction step*, we show that we can extend this one more step to a prefix of length $l + 1$. Given the prefix of $\vec{w_\forall}$ with length $l + 1$, i.e., we have some $\vec{u^l} \cdot \vec{v} = \vec{w_\forall}[0] \dots \vec{w_\forall}[l]$ with $\vec{u^l} \in (\Upsilon^n)^l$ and $\vec{v} \in \Upsilon^n$. Calculate $\sigma(\vec{u^l} \cdot \vec{v})$. For that, $\sigma'(\vec{u^l}) = q$ is computed, as well as $\sigma'(\vec{u^l} \cdot \vec{v})$. During the computation of the latter, inputs $\vec{v'} \in \Upsilon^m$ to extend the existentially quantified traces and a corresponding successor state $q' \in Q^\times$ for which $p^{q'}$ holds are found. We know that these $\vec{v'}$ and q' exist so that the default option

of σ' is not used, because the induction hypothesis and the correctness of the prophecy variables' previous guesses mean that there has to be at least one choice for $\vec{v'}$ that leads to a successor state that lies on the existing accepting run. The function σ' returns one appropriate successor state because also in the current step the prophecy variables guess correctly by assumption and, thus, at least one $p^{q'}$ has to be *true*. This prophecy variable points to a successor state q' that lies on an accepting run for the predicted future behavior of the universally quantified traces. Thus, the corresponding input $\vec{v'}$ that σ returns does not violate the formula because it does not cause a bad prefix. This is known because there is an accepting run where q' is a successor state. Because σ avoids causing a bad prefix in every step, we can conclude that nothing bad ever happens. Thus, $\vec{w_\forall}$ together with $\vec{w_\exists}$ defined by σ result in a trace ρ that satisfies the formula and the run created by σ' is an accepting run in $\mathcal{S}^{n+m} \times MH(\mathcal{A}_\varphi)$. So, we can conclude that the strategy σ is winning.
□

Appendix B

Reviews

In this chapter, the reviews[1] that we received are printed. First, the reviews to the paper are given before the changes that we made for the camera-ready version are described. Then, the reviews from the artifact evaluation committee follow.

B.1 Reviews to the Paper

We received three reviews for the paper and a Metareview summarizing the three main reviews. All of the reviewers were in favor of accepting the paper to CAV'19. The overall evaluation of two of the reviews was 'accept' (total score 2) and the third review voted for a 'weak accept' (total score 1).

- **Review 1: Overall evaluation: 2 (accept)**

  The paper addresses the problem of verifying that a finite-state system satisfies a HyperLTL property with a single quantifier alternation: $\forall^*\exists^*$ or $\exists^*\forall^*$. To verify such properties while avoiding the (known) high complexity, the authors suggest an incomplete reduction to verification of universal HyperLTL properties. The reduction replaces the existential path quantifiers with a strategy for computing the corresponding paths. If the strategy is given, verification reduces to the much simpler task of verifying universal properties. To improve the applicability of the reduction (which is in general incomplete, i.e., a system may satisfy the property even though no strategy exists), the authors consider augmenting the system with prophecy variables.

[1]All reviews are printed without major changes. The only changes that were made address the presentation of the reviews to increase the readability including formatting, adding LaTeX-math mode, correcting the capitalization and obvious typos, e.g., *talsk* to *talks*. The content is, however, unchanged.

To find strategies, the authors suggest a bounded synthesis approach based on encoding the problem of finding a strategy via logical constraints.

Finally, the authors also consider synthesis of a system that satisfies a given HyperLTL formula with one quantifier alternation. Synthesis is also based on the concept of strategies and essentially decomposes synthesis for properties with one quantifier alternation to synthesis of a system for a quantifier-free property together with a strategy.

The verification approach has been implemented in MCHyper, where the strategy is manually provided. The authors consider a few examples that were previously verified by manually approximating the property, and show that their approach is comparable in terms of running time.

Synthesis was implemented in a separate tool. As expected due to the high complexity, it manages to synthesize only tiny programs (up to 3 states).

The paper is well written and I enjoyed reading it. It brings together several known ideas from other contexts to obtain a new approach for tackling verification as well as bounded synthesis of hyperproperties with quantifier alternations. It makes a significant theoretical contribution in the area of verifying HyperLTL properties. The practical contribution is less clear, since the approach doesn't seem to scale very well (the verification problems have at most 90 latches, and do not even include the synthesis of the strategy, and the synthesis problems have at most 3 states).

Small comments:

- The use of strategies instead of existential quantifiers (including prophecy variables) resembles ideas used in verification of CTL*, e.g.:

  Byron Cook, Heidy Khlaaf, Nir Piterman: On Automation of CTL* Verification for Infinite-State Systems. CAV (1) 2015: 13-29
- There are some clashes in the notations. For example, in page 5, τ first denotes the transition function of a transition system, and later denotes a node in a tree. In page 5, def of transition systems, it is worth noting that you assume AP = I ∪ O (as the definition of HyperLTL talks about AP). In page 6, τ^* denotes the transition function of the composition of a transition system with a strategy. This clashes with the extension of τ to sequences.
- Isn't verification of $\exists^*\forall^*$ properties easier than verification of $\forall^*\exists^*$ properties? Some discussion seems in place.

  - Thm 4: please say that σ is a finite-state strategy (or say earlier that all strategies you consider are finite-state). Otherwise, the composition with σ is not well defined.
  - p9 "Towards Completeness": You say "Our completeness result is restricted to finite-state systems..." – my understanding was that the entire paper is formulated for finite state transition systems (this is the definition in the preliminaries). If you had in mind a more general setting, it is worth stating explicitly.
  - Thm 6: Please state the extension of HyperLTL more formally.
  - p10-p11 "Bounded Synthesis of Strategies" – You switch between φ and ψ in an inconsistent manner.
  - Thm 8: Please define the acceptance condition. I assume it is inherited from the property automaton, and is therefore a co-Büchi condition. In fact, where did you use a parity acceptance condition in the paper?
  - The definition before Lemma 1 could use some explanation. I first read the definition of the initial vertices as requiring that for every v, init(v)=t (for the same t). I suggest rephrasing to avoid such confusion.

- **Review 2: Overall evaluation: 1 (weak accept)**

  This paper introduces an algorithm for verifying HyperLTL formulas. HyperLTL is an extension of LTL that allows quantifying over program traces, which allows the expression of hyperproperties (standard LTL does not allow reasoning across multiple program traces). Previous work supports only HyperLTL formulas expressed without quantifier alternation e.g. properties involving only forall-quantifiers, which correspond to hypersafety properties. A transformation exists for reducing all properties to this subclass, but it is prohibitively expensive. A game-theoretic solution is proposed for solving formulas with a single quantifier alternation: Validity is implied by the existence of a strategy for selecting existential witnesses given universal choices. This initial solution is deliberately incomplete in order to remain tractable: specifically, the existential player is only allowed to view past choices of the universal player. This limitation is partially worked around by adding prophecy variables to the input formulas, which allow bounded lookahead. It is shown how to automatically synthesize a strategy and appropriate prophecy variables (up to a user-given bound on lookahead) by encoding the problem as an SMT instance. Finally, the algorithm is extended to support arbitrary quantifier alternation.

In summary, the paper addresses an important and well-motivated problem and presents a nice solution that advances the state of the art. But, there are problems with the presentation.

The title is misleading. The proposed solution does not work for just hyperliveness properties, but for all hyperproperties. I recommend the authors change the title to Quantified HyperLTL Verification, to better reflect the core contribution of the paper in the title.

The paper is nicely structured and I believe I understand the general idea (which I find clean and well-motivated). The introduction is the worst written part of the paper. I had to look up 3 papers to fully understand what was going on with the properties used as an example in the introduction. A little more effort, and a mention of nondeterminism as the source of the problem would ease the reader into the context much better.

More importantly, there are a few errors in the technical part of the paper which hinder forming an in-depth understanding of the specifics - particularly the encoding of the problem as an SMT instance. These are:

1. The definition of $\uplus$ is given as $(\mathrm{x} \uplus \mathrm{y})[\mathrm{i}] = \mathrm{x}[\mathrm{i}] \cup \mathrm{y}[\mathrm{i}]$, but x and y are not defined to be vectors of sets.
2. The definition of trees given at the end of page 5 is unclear. Are these trees finite or infinite? Condition (ii) is given for every node τ and every positive integer n. Does this not imply that every node has an infinite number of children? If so the introduction of m is unnecessary, since m effectively ranges over every positive integer as well.
3. There is a minor error in the definition of transition systems on page 5: $\tau^*(\epsilon)$ should be defined as t_0, not q_0.

Finally, the set of benchmarks is very small and does not instill confidence that the solution performs well on more diverse examples. If tractability is the main issue for this problem space, one cannot conclude that that problem has been sufficiently solved based on the experimental results presented in the paper.

Related to this last point, consider a property like linearizability. At least on the surface, it looks like a $\forall\exists$ type property (i.e. for all paths, there exists an equivalent sequential path). Why not take a property like this which is more familiar than the rest, and show that the algorithm performs well?

If this paper were well-written, it would be a clear accept for me. Under these conditions (problems with the writing and the extent of experimentation), I am on the fence about it.

- **Review 3: Overall evaluation: 2 (accept)**

  The paper considers model checking HyperLTL properties with one quantifier alternation (either $\forall\exists$ or $\exists\forall$). The proposed technique uses a strategy implemented by a transition system as a witness for the existential quantifiers, which reduces the problem to checking a universally quantified formula (similar to Skolemization). Using a strategy raises an issue of incompleteness, which the authors explore and address using prophecy variables. The paper considers both user provided strategies and automatically synthesizing strategies. The paper also considers automatically synthesizing reactive systems based on HyperLTL specifications with quantifier alternations.

  The paper considers a well-defined problem and seems to have a clear contribution beyond the state-of-the-art. The suggestion to use a strategy as a witness for existential quantifiers is appealing and natural, and the nice issue of incompleteness that it raises is explored by the paper. I also liked Theorem 6, which clearly explains what is needed for completeness. I found the paper clear and well-written. Below, I list a few points that were less clear, and more minor presentation issues.

  One point that was not clear enough for me was the restriction to one quantifier alternation. It seems that some parts of the paper assume one quantifier alternation, while others do not take this restriction. My understanding is that the evaluation included only formulas with one quantifier alternation. However, it seems that the proposed technique could in principle be applied to any quantifier structure, same as Skolemization in first-order logic. Theorems 4 and 5 seem to easily generalize to arbitrary quantifiers, and it is intriguing if Theorem 6 generalizes as well. Either way, the paper can be improved by addressing this more clearly.

  The evaluation section spends most of the discussion about model checking with given strategies, while I think the more interesting part is the one where the witness strategy is synthesized. If the strategy is manually provided, the difference from model checking of universal HyperLTL properties seems superficial.

  A missing reference is:
  Finkbeiner B., Hahn C., Hans T. (2018) MGHyper: Checking Satisfiability of HyperLTL Formulas Beyond the $\exists^*\forall^*$ Fragment. ATVA 2018. (`https://doi.org/10.1007/978-3-030-01090-4_31`).
  While this tool paper uses a different technique compared to the submission, it seems to solve the same problem: Model checking HyperLTL with quantifier alternations. It would also be interesting to compare to this tool in the evaluation.

Page 7, before Theorem 3: "... more difficult and, thus, intractable in practice". I find it problematic to say that a problem is intractable in practice *because* it is EXPSPACE-complete, while a PSPACE-complete problem is regarded as tractable. The complexity classes are important to understand, but they do not tell the full story of "tractable in practice".
Page 7, before Section 3: "we present a technique that solves the complexity problem" – rephrase to be more accurate.

- **Metareview:** The reviewers found the paper to be well-motivated and to make a significant contribution to the state of the art, despite some concern about scalability and generality of the method. Though there were some issues concerning clarity of the presentation, the reviewers feel that these can be addressed.

B.1.1 Changes Made for the Camera-Ready Version

This thesis is based on the model checking parts of the final version of the paper "Verifying Hyperliveness" [15]. In this thesis, we have adapted the presentation of the results and extended the description of the experiments giving much more details about the experimental evaluation for MCHyper.
For the final version, we made some changes to the paper. Following the reviewers' suggestions, we fixed the issues with the presentation for the camera-ready version. In particular, we have improved the introduction giving more intuition and context about the HyperLTL formulas used as examples and we have fixed the inconsistencies and typos found by the reviewers.
Moreover, we omitted the paragraph "Towards Completeness". Even though Reviewer 3 liked our exploration of the incompleteness of our approach we had to agree with Reviewer 1 that the presentation of the necessary extension needed to be improved. While we removed this part for the final version of the paper, we have included an improved version of it in this thesis.

B.2 Reviews to the Artifact

We submitted a virtual machine containing our experimental data to the CAV artifact evaluation committee. This was not required for regular papers but we were invited to submit our artifact for evaluation. If the evaluation is successful, i.e., if the artifact evaluation committee can reproduce the results of the experiments reported in the paper using the provided virtual machine and the artifact was well documented, then the paper is allowed to use a seal that shows the successful artifact evaluation.
We received three reviews to our artifact, all with a total score of 2 (accept). The final version of our paper [15] therefore uses the artifact evaluation seal.

In the following, we print the reviews to our artifact containing the extension of MCHyper and the extension of the bounded synthesis tool BoSy.

- **Review 1: Total score: 2 (accept)**

  Easy to reuse.

  - The provided artifact has README.md for running benchmarks and the virtual machine has all the running environment configured.
  - It would be helpful to provide an instruction of installing virtual machine.

  Consistent.

  - The actual running time measured by running the artifact on the provided virtual machine is smaller than the time reported on paper, and this has been mentioned in the README of this artifact. It is better to provide some explanations for this case.
  - The reported time of running these benchmark has three different time metrics (real, user, sys), it is better to give instruction on choosing which time metrics to add/sum.

  Complete.

  - All the benchmark results of execution time match the expected range declared by the author and consistently smaller than reported results in the paper.

  Well documented.

  - The artifact is well documented with README to building and running programs.
  - I didn't find the demonstration of applying the presented method to new input.

- **Review 2: Total score: 2 (accept)**

  Summary

  This paper studies hyperliveness properties expressed as HyperLTL formulas with quantifier alternation. I evaluated the two tools in the provided VM, namely an extension of MCHyper and a separate bounded synthesis tool with z3. I'm running the evaluation in a VM with 8G of RAM, i5 2.9GHz (host), 1 core (default setting of the VM).

  Reusability

  The artifact is easy to reuse. The provided readme files contain tool building instructions and running instructions. I follow the running instructions and the two experiments run successfully within the given/expected time.

Consistency and Completeness
I can reproduce most of the reported results from the paper consistently and completely, but it seems that a few experiments from the MCHyper take slightly more time than reported:
Line 5 in Table 1 (reported $50.6s$)
Try 1: $55.058s$
Try 2: $57.700s$
Try 3: $49.709s$
Line 6 in Table 1 (reported $27.5s$)
Try 1: $37.204s$
Try 2: $31.481s$
Try 3: $34.089s$
Line 8 in Table 1 (expected $< 8min$)
Try 1: $7m12.627s$
Try 2: $8m8.870s$
Try 3: $8m45.136s$
The above are results collected from three separate runs. Specifically for Line 6, the time costs from all the three separate runs are consistently more than $27.5s$ in the paper.
The BoSy-cav experiment sets perform better than reported, as indicated in the README files provided.
Documentation
The documentation is detailed and easy to follow. The artifact also provides further options of the experiment.

- **Review 3: Total score: 2 (accept)**
  The provided artifact is reasonably easy to reuse following the included documentation and scripts. The provided scripts were very helpful in reproducing the results in the tables, and I could reproduce all the results in the tables. As mentioned in the MCHyper README, the times reported on the VM for the MCHyper results were consistently faster than the ones in the paper. How to apply the MCHyper tool to new inputs is clearly documented in the corresponding README. How to apply the BoSyHyper tool to new inputs is not as clearly documented, though I believe examining the provided script for reproducing the synthesis results would allow one to apply the tool to new inputs.