# Complexity of Model Checking Second-Order Hyperproperties on Finite Structures

Saarland University

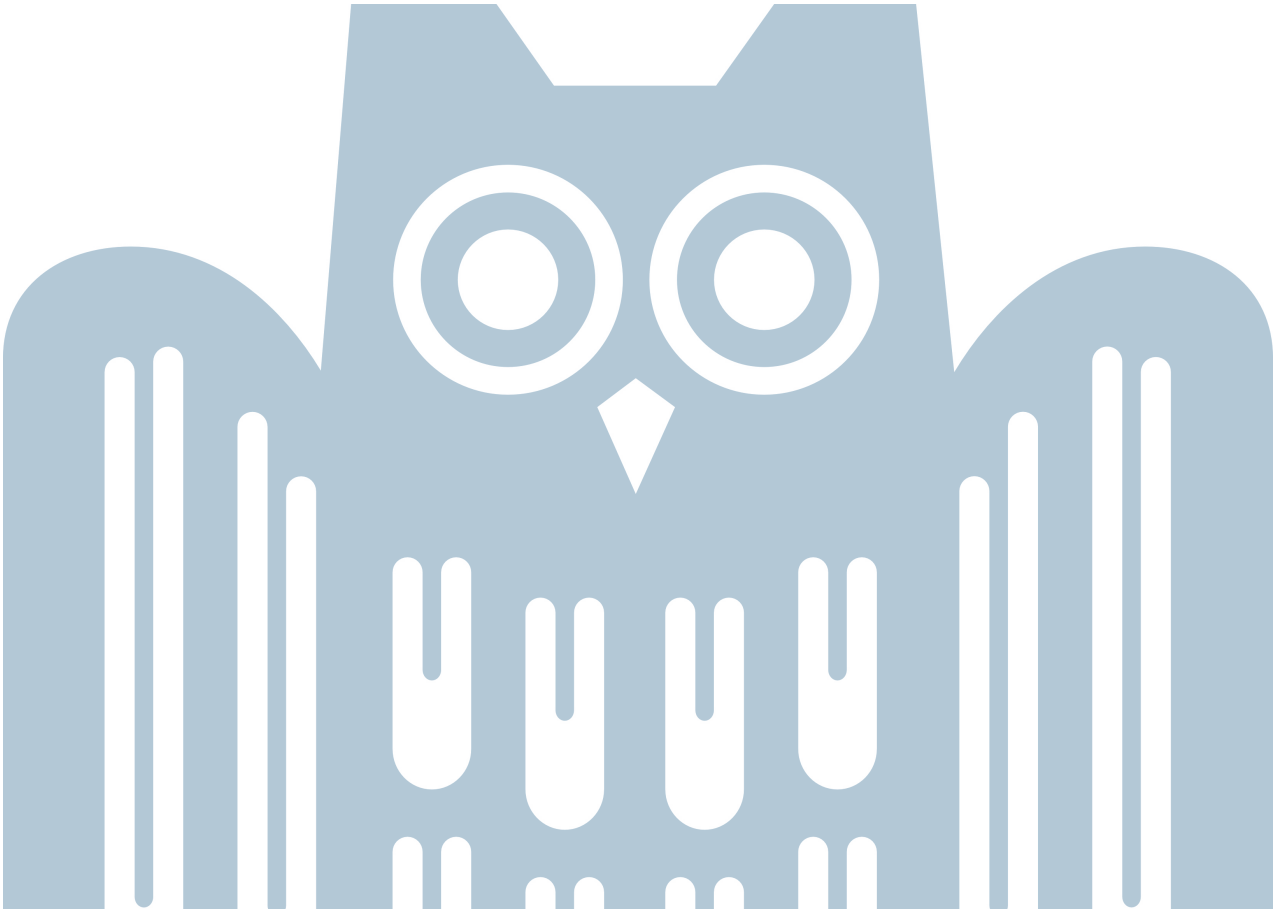
Department of Computer Science

Bachelor's Thesis

*submitted by*

Tim Henri Rohde

Saarbrücken, October 2023



| | |
|---|---|
| Supervisor: | Prof. Bernd Finkbeiner, Ph.D. |
| Advisor: | Prof. Bernd Finkbeiner, Ph.D. |
| | Dr. Hadar Frenkel |
| Reviewer: | Prof. Bernd Finkbeiner, Ph.D. |
| | Dr. Hadar Frenkel |
| Submission: | October 20, 2023 |

## Abstract

Model checking is an automatic approach to verification. For model checking the system (model) is given as Kripke structure and the specification is written in a temporal logic. Hyper$^2$LTL is a new temporal logic that extends HyperLTL by the quantification over sets of traces. It can express second-order hyperproperties, such as common knowledge. Since model checking Hyper$^2$LTL is in general undecidable, we restrict the models to be acyclic and analyze the complexity in the size of the model. We show that Hyper$^2$LTL model checking is decidable on acyclic models. It is in PSPACE on tree-shaped models and in EXPSPACE on acyclic models. Additionally, we show that for a powerful fragment of Hyper$^2$LTL, called Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$, model checking is P-complete on tree-shaped models and EXP-complete on acyclic models.

## Acknowledgements

First of all, I would like to thank Prof. Finkbeiner and Hadar, for offering me this thesis. Their advice was very helpful and they always had time for me although they had a lot of other things to do.

Furthermore, I would like to thank Jonathan for patiently answering all my questions regarding complexity and thank you to Iona and Benjamin for answering all my other questions.

Finally, I would like to thank Lea for always supporting me and making sure I had enough sleep, food and Cola.

**Eidesstattliche Erklärung**
Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

**Statement in Lieu of an Oath**
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

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

**Declaration of Consent**
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.

---

Saarbrücken, 20 October, 2023

**Erklärung**
Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

**Statement**
I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

---

Saarbrücken, 20 October, 2023

# Contents

# 1 Introduction

With digital technologies becoming more and more involved in critical situations, incorrect systems become more dangerous. One automatic approach to ensure the correctness of a system is *model checking*, where a program and a specification are given and it is automatically checked whether the program satisfies the specification. The program, in this context usually called *model*, is given as a finite transition system. Such a transition system contains all executions the system can make in the form of traces. A *trace* is a sequence of steps that are annotated with conditions that hold in the corresponding step. The specifications can be expressed in different *temporal logics*. Temporal logics specify properties that the set of traces of a system has to satisfy. The model checking problem of a temporal logic is to decide whether a system satisfies a specification given in that temporal logic. The more expressive such a temporal logic is, the more complex or even undecidable becomes its model checking problem.

Many properties that specify the correctness of a system only need to reason over one execution. This includes statements like "nothing bad happens" or "the program eventually makes progress" but also that the output of a system always satisfies some condition.

The correctness of systems that maintain sensitive information often has to be expressed as hyperproperties to specify who can derive facts about this information. Hyperproperties relate several execution traces with each other and can therefore reason about information flow. One interesting example of a hyperproperty is knowledge:

**Example 1.0.1** (Knowledge)**.** Let $\mathsf{P}$ be a program containing public and secret variables and let $\mathsf{A}$ be an agent observing the public variables on one execution $\pi$ of the program. $\mathsf{A}$ knows $\varphi$ if $\varphi$ holds on all traces in $\mathsf{P}$ that $\mathsf{A}$ connot distinguish from $\pi$ by looking at public variables. △

Hyperproperties can also express other properties involving information flow, like noninterference.

HyperLTL is a temporal logic that can express hyperproperties. It extends the temporal logic LTL by the quantification over traces [Cla+14]. The complexity of the model checking problem for HyperLTL is very well analyzed on finite models as well as on acyclic and tree-shaped models [BF18].

While the complexity of model checking hyperproperties is well analyzed with HyperLTL, it is not well analyzed for second-order hyperproperties. Second-order hyperproperties are properties that relate different sets of traces. One interesting second-order

hyperproperty is common knowledge. A fact $\varphi$ is common knowledge within a group of agents if every agent knows $\varphi$, every agent knows that every agent knows $\varphi$, every agent knows that every agent knows that every agent knows $\varphi$ and so on, building an infinite chain of knowledge. To express common knowledge, a temporal logic has to be able to reason over arbitrary sets of traces, therefore it is a second-order hyperproperty [BMP15]. Common knowledge is of particular interest if several distributed agents need to be coordinated, for example, if they have to act simultaneously.

Hyper$^2$LTL is a temporal logic that is able to express second-order hyperproperties. It extends HyperLTL by second-order quantification, i.e. the quantification over sets of traces. If the model is finite but not restricted further, the Hyper$^2$LTL model checking problem is undecidable [Beu+23].

An interesting fragment of Hyper$^2$LTL is Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$. The fragment restricts the second-order quantification to sets that are defined by a given fixpoint formula. This only allows quantification over exactly one set per second-order quantifier, which makes the model checking problem less complex. In spite of this, Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ is still able to express many interesting second-order hyperproperties including common knowledge. Unfortunately, its model checking problem is still undecidable. Beutner et al. present an approximate model checking algorithm for Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ on finite models [Beu+23]. We study in this thesis the Hyper$^2$LTL model checking problem as well as the Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ model checking problem on acyclic models.

Such acyclic models occur if the system's structure is acyclic (e.g. it terminates) or when monitoring a system. Monitoring a system means that the structure of the system is not known and we can only observe traces that are in the system. For this, all occurring traces are collected and composed into a model which is then the input to a model checking algorithm. If the traces are ordered by common prefixes, then a tree-shaped model is obtained, if the traces are ordered by common pre- and suffixes, an acyclic model is obtained. Additionally, for monitoring, the behavior of the model checkers' runtime is only relevant in the size of the structure. This is because the model grows with every program execution but the specification does not. Therefore, all complexities given in this thesis are in the size of the model, except when mentioned otherwise.

In this thesis, we do complexity analysis under the assumption that only the model is part of the input and the formula is of constant size. The upper bounds are shown by providing model checking algorithms. The lower bounds are shown by reducing from other problems in the respective complexity class.

The complexity results of this thesis do not only provide model checking algorithms but the given lower bounds also prove that there is no model checking algorithm that is asymptotically faster. Additionally, we get a good idea of how expensive the analyzed model checking problem is in comparison to other model checking problems.

| | Tree-shaped | Acyclic |
|---|---|---|
| Fixpoint Hyper$^2$LTL$_{fp}$ | P-complete (Thm. 4) | EXP-complete (Thm. 14) |
| Hyper$^2$LTL $(\exists\forall)^k$ | $\Sigma^P_{k+1}$-complete (Thm. 18) | $\Sigma^{EXP}_{k+1}$ (Lem. 20) |
| Hyper$^2$LTL $(\forall\exists)^k$ | $\Pi^P_{k+1}$-complete (Thm. 18) | $\Pi^{EXP}_{k+1}$ (Lem. 20) |
| Hyper$^2$LTL | PSPACE (Cor. 19) | EXPSPACE (Cor. 21) |

Figure 1.1: The complexity of Hyper$^2$LTL model checking in the size of the Kripke structure. $(\exists\forall)^k$ and $(\forall\exists)^k$ denote that the problem is restricted to formulas with $k$ second-order quantifier alternations.

A summary of the results from this thesis can be seen in Fig. 1.1. We show that for Fixpoint Hyper$^2$LTL$_{fp}$ the model checking problem is P-complete if the model is restricted to be tree-shaped (Thm. 4) and EXP-complete if the model is restricted to be acyclic (Thm. 14). The lower bound proofs reduce from Horn-satisfiability for the P-completeness proof and from the Succinct Circuit Value Problem for the EXP-completeness proof.

→ Thm. 4, p. 21
→ Thm. 14, p. 40

To analyze the complexity of the full logic, we distinguish the number of second-order quantifier alternations in the given formula. We show that model checking Hyper$^2$LTL on tree-shaped structures is $\Sigma^P_{k+1}$-complete if the formula has $k$ second-order quantifier alternations and its outermost second-order quantifier is existential. For formulas where the outermost second-order quantification is universal, the model checking problem is $\Pi^P_{k+1}$-complete if the formula has $k$ second-order quantifier alternations (Thm. 18).

→ Thm. 18, p. 50

When considering acyclic Kripke structures, we show that the problem is contained in $\Sigma^{EXP}_{k+1}$ or $\Pi^{EXP}_{k+1}$ where $k$ again denotes the number of second-order quantifier alternations. The problem is in $\Sigma^{EXP}_{k+1}$ if the outermost second-order quantifier in the given formula is existential and in $\Pi^{EXP}_{k+1}$ otherwise (Lem. 20).

→ Lem. 20, p. 51

Finally, we show that MC[Fixpoint, Tree] as well as MC[Hyper$^2$LTL, Tree] is PSPACE-complete in the combined input consisting of Kripke structure and formula (Thm. 23, Thm. 24).

→ Thm. 23, p. 52
→ Thm. 24, p. 52

The remainder of this thesis is structured as follows: First, we discuss related work in Chapter 2. Then, we define LTL, HyperLTL, Hyper$^2$LTL and its fragment in Chapter 3. Further, we give formal definitions of the used complexity classes there. In Chapter 4 we analyze the complexity of the model checking problem for Fixpoint Hyper$^2$LTL$_{fp}$ and in Chapter 5 we continue with the analysis of the model checking problem for full Hyper$^2$LTL. We discuss the results and open questions in Chapter 6.

# 2 Related Work

The model checking problems of less complex logics are very well analyzed. For most temporal logics there exist complexity analysis as well as model checking algorithms.

## 2.1 Complexities of Other Logics

There are already complexity analysis for logics that are subsumed by Hyper$^2$LTL. Bonakdarpour and Finkbeiner [BF18] analyzed the complexity of the model checking problem for HyperLTL on trees and acyclic graphs. They showed that the complexity of the model checking problem for HyperLTL shrinks significantly when the model is restricted to be acyclic or tree-shaped. This motivates our work analyzing the complexity of the model checking problem for Hyper$^2$LTL to see if it becomes decidable on acyclic structures.

$\text{LTL}_{K,C}$ and A-HLTL are two extensions of LTL whose model checking problems are undecidable in general and which are subsumed by Hyper$^2$LTL. $\text{LTL}_{K,C}$ extends LTL by operators to express knowledge and common knowledge. Its model checking problem is also undecidable in general [MS99]. However, model checking $\text{LTL}_{K,C}$ becomes decidable as soon as the temporal operators are removed [MS99].

A-HLTL extends HyperLTL to express asynchronous hyperproperties. Just like $\text{LTL}_{K,C}$ its model checking problem is undecidable in general but becomes decidable when restricting the logic or model. Baumeister et al. [Bau+21] identify some of the fragments by restricting the syntactic form of the formula and Hsu et al. [Hsu+23] show that model checking A-HLTL is decidable on acyclic graphs. Because we focus on restricting the given structure to be acyclic or tree-shaped especially the second result hints to the fact that at least some of the model checking problems we analyze in this thesis are decidable.

## 2.2 Model Checking Algorithms

In this thesis, we give several model checking algorithms. Most of them simply enumerate all possible quantifier instantiations which is sufficient to show upper bounds but there are many model checking algorithms that use more elegant tactics. Many of these algorithms are based on automata theory, for example, the usual LTL model checking algorithm [BK08]. Another example is the Hyper$^2$LTL$_{fp}$ model checking algorithm that uses automata to maintain sets of traces [Beu+23]. A different approach

is to reduce the model checking problem to a problem for which algorithms already exist. Baumeister et al. [Bau+21] for example reduce one of the analyzed problems to HyperLTL model checking. Another approach to model checking are bounded model checking algorithms [Hsu+23]. Bounded model checking algorithms only look at traces with a bounded length and are not necessarily sound on general graphs, but are sound on acyclic graphs.

## 2.3 Finite Traces

In acyclic and tree-shaped structures, all traces end in a self-loop such that LTL semantics for finite traces would be applicable. There are many definitions for LTL semantics for finite traces, many of which exist in a strong and a weak version. For example, in monitoring, strong and weak semantics are often defined such that a trace is only accepted if all possible continuations resp. one possible continuation satisfy the formula [KV99]. Another example would be [Eis+03] which defines strong semantics such that no formula is satisfied after the end of the finite prefix and weak semantics such that all formulas hold after the prefix ends.

Using one of these semantics would require us to change the HyperLTL and Hyper$^2$LTL semantics. Since they are defined over Kripke structures instead of traces, this change would raise other choices. Additionally, reasoning over all possible continuations of a trace is more complex than reasoning over one infinite trace such that the complexity of the model checking problem for Hyper$^2$LTL would most likely change if we use such finite LTL semantics.

# 3 Preliminaries

## 3.1 Models

We start by defining Kripke structures, which are the given models.

**Definition 3.1** (Kripke structure)

Let $AP$ be a set of atomic propositions. A *Kripke structure* is a tuple $(S, s_0, \delta, L)$ such that

- $S$ is a finite set of states;
- $s_0 \in S$ is the initial state;
- $\delta \subseteq S \times S$ is the transition relation and
- $L : S \mapsto 2^{AP}$ is a labeling function.

It is required that $\forall s \in S.\delta(s) \neq \emptyset$.

We call a Kripke structure *acyclic* if the underlying graph $(S, \delta)$ is acyclic. States that have no outgoing transition are allowed to have a self-loop.

**Def.** Acyclic Kripke structure

We call a Kripke structure *tree-shaped* if and only if for every state $s \neq s_0$ there exists exactly one state $s' \neq s$ such that $(s', s) \in \delta$. $s_0$ is not allowed to have any predecessors. States that have no successor are allowed to have a self-loop.

**Def.** Tree-shaped Kripke structure

Kripke structures contain all relevant traces, which are defined as follows:

**Definition 3.2** (Trace)

Given a set of atomic propositions $AP$, a *trace* $\pi \in (2^{AP})^\omega$ is an infinite word over the alphabet $2^{AP}$.

Let $\pi$ be a trace. We write $\pi[i]$ for the set at position $i + 1$ of $\pi$ and we write $\pi[i, \infty]$ for the suffix of $\pi$ starting at position $i + 1$.

In this thesis, we are only dealing with traces that end in a self-loop after finitely many steps. In several reductions, we encode binary numbers on these traces. We say that some trace $t$ has binary number or bitstring $b$ encoded with proposition $p$ on them iff $p$ holds in the $i$-th step of $t$ if and only if $b$ has a 1 at position $i$. Given two traces $t$ and $t'$ of equal length [1] encoding some number with proposition $p$ then the formula $\square(p_t \leftrightarrow p_{t'})$ is satisfied if and only if $t$ and $t'$ encode the same number.

[1] e.g. they end in a self-loop after equally many steps

## 3.2 HyperLTL

The syntax of HyperLTL [Cla+14] is as follows:

$$\varphi ::= \forall \pi.\varphi \mid \exists \pi.\varphi \mid \psi$$

$$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \psi \,\mathcal{U}\, \psi \mid \bigcirc \psi \quad a \in AP$$

where $AP$ is a set of atomic propositions and $\pi \in \mathcal{V}$ where $V$ is a set of trace variables.

The semantics of HyperLTL are defined over the relation $\models_T$. $T$ is a set of traces, usually given as Kripke structure. $\models_T$ relates trace assignments to HyperLTL formulas. A trace assignment is a function $\Pi : \mathcal{V} \rightharpoonup T$. We write $\Pi[i, \infty]$ for the assignment $\Pi'(\pi) = \Pi(\pi)[i, \infty]$ and $\Pi[\pi \mapsto t]$ for the assignment that is equal to $\Pi$ but maps $\pi$ to $t$.

| | |
|---|---|
| $\Pi \models_T a_\pi$ | iff $a \in \Pi(\pi)[0]$ |
| $\Pi \models_T \neg\psi$ | iff not $\Pi \models_T \psi$ |
| $\Pi \models_T \psi_1 \vee \psi_2$ | iff $\Pi \models_T \psi_1$ or $\Pi \models_T \psi_2$ |
| $\Pi \models_T \bigcirc \psi$ | iff $\Pi[1, \infty] \models_T \psi$ |
| $\Pi \models_T \psi_1 \,\mathcal{U}\, \psi_2$ | iff there exists $i \geqslant 0 : \Pi[i, \infty] \models_T \psi_2$ and for all $0 \leqslant j < i$ holds $\Pi[j, \infty] \models_T \psi_1$ |
| $\Pi \models_T \forall \pi.\varphi$ | iff for all $t \in T : \Pi[\pi \mapsto t] \models_T \varphi$ |
| $\Pi \models_T \exists \pi.\varphi$ | iff there exists $t \in T : \Pi[\pi \mapsto t] \models_T \varphi$ |

Additionally, we define the usual derived notations. Apart from the Boolean operators, this includes finally / eventually $\Diamond$, globally $\Box$, equality between traces and exists exactly one $\exists!$.

$$
\begin{aligned}
\mathit{true} &\equiv a \vee \neg a \qquad \text{for some } a \in AP \\
\mathit{false} &\equiv \neg \mathit{true} \\
\psi_1 \wedge \psi_2 &\equiv \neg(\neg\psi_1 \vee \neg\psi_2) \\
\psi_1 \rightarrow \psi_2 &\equiv \neg\psi_2 \vee \psi_1 \\
\psi_1 \leftrightarrow \psi_2 &\equiv (\psi_1 \rightarrow \psi_2) \wedge (\psi_2 \rightarrow \psi_1) \\
\Diamond \psi &\equiv \mathit{true}\, \mathcal{U}\, \psi \\
\Box \psi &\equiv \neg \Diamond \neg \psi \\
\pi =_{AP} \pi' &\equiv \bigwedge_{a \in AP} \Box(a_\pi \leftrightarrow a_{\pi'}) \\
\exists! \pi. \varphi(\pi) &\equiv \exists \pi. \forall \pi'. \varphi(\pi) \wedge (\varphi(\pi') \rightarrow \pi =_{AP} \pi')
\end{aligned}
$$

If $K$ is a Kripke structure and $\mathit{Traces}(K)$ denotes all traces in $K$, then we usually write $K \models \varphi$ instead of $[] \models_{\mathit{Traces}(K)} \varphi$.

## 3.3 Hyper$^2$LTL

Hyper$^2$LTL was introduced by [Beu+23] and extends HyperLTL by second-order quantification over sets of traces. Its syntax introduces two new quantifiers over sets and the first-order quantifiers specify a set the traces come from:

$$
\begin{aligned}
\varphi &::= \forall X. \varphi \mid \exists X. \varphi \mid \forall \pi \in X. \varphi \mid \exists \pi \in X. \varphi \mid \psi && \pi \in \mathcal{V}, X \in \mathfrak{V} \\
\psi &::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \psi\, \mathcal{U}\, \psi \mid \bigcirc \psi && a \in AP
\end{aligned}
$$

In addition to the set of trace variables $\mathcal{V}$ we have a set of second-order variables $\mathfrak{V}$.

The $\models$ relation for Hyper$^2$LTL relates pairs consisting of a trace assignment and a second-order assignment to Hyper$^2$LTL formulas. A second-order assignment is a partial function $\Delta : \mathfrak{V} \rightharpoonup 2^T$. The semantics of Hyper$^2$LTL are then defined as follows:

$$
\begin{aligned}
&\Pi, \Delta \models \psi && \text{iff } \Pi \models_T \psi \\
&\Pi, \Delta \models \forall \pi \in X. \varphi && \text{iff for all } t \in \Delta(X) : \Pi[\pi \mapsto t], \Delta \models \varphi \\
&\Pi, \Delta \models \exists \pi \in X. \varphi && \text{iff there exists } t \in \Delta(x) : \Pi[\pi \mapsto t], \Delta \models \varphi \\
&\Pi, \Delta \models \forall X. \varphi && \text{iff for all } A \subseteq T : \Pi, \Delta[X \mapsto A] \models \varphi \\
&\Pi, \Delta \models \exists X. \varphi && \text{iff there exists } A \subseteq T : \Pi, \Delta[X \mapsto A] \models \varphi
\end{aligned}
$$

We say that a Kripke structure $K$ containing traces $\mathit{Traces}(K)$ satisfies Hyper$^2$LTL formula $\varphi$ if and only if $\emptyset, [\mathfrak{G} \mapsto \mathit{Traces}(K)] \models \varphi$. In this case, we instead write $K \models \varphi$.

We define the same syntactic sugar as for HyperLTL on Hyper$^2$LTL. Further, we define the following syntactic sugar for Hyper$^2$LTL:

$$\pi \triangleright X \equiv \exists \pi' \in X. \square(\pi' =_{AP} \pi)$$

$\pi \triangleright X$ expresses that trace $\pi$ is contained in set $X$. This syntactic sugar contains a first-order quantifier but as long as $\triangleright$ is not used under temporal operators, the formula can be transformed into valid Hyper$^2$LTL syntax.

At several points in this thesis, we case distinct over the number of second-order quantifier alternations, which are defined as follows:

**Definition 3.3** (Second-Order Quantifier Alternations)

The number of second-order quantifier alternations of a Hyper$^2$LTL formula $\varphi$, is the number of times $\varphi$ changes from universal to existential second-order quantification or vice versa. All first-order quantifications are ignored.

**Example 3.3.1.** The formula $\exists A. \forall \pi \in A. \exists B. \forall C. \exists D. \ldots$ has two second-order quantifier alternations. One second-order quantifier alternation from existential to universal and one second-order quantifier alternation back to existential quantification. △

The semantics we use differ from the original Hyper$^2$LTL semantics introduced in [Beu+23]. They use a second special second-order variable $\mathfrak{U}$ to represent the set of all possible traces, even those not contained in the given Kripke structure. Because we analyze the model checking problem for restricted models, semantics that can argue over all possible traces are not suited for our approach.

## 3.4 Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$

The fragment of Hyper$^2$LTL we are analyzing is Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$. It is introduced with Hyper$^2$LTL as a powerful fragment that is less complex to model check [Beu+23].

Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ restricts the second-order quantification to the quantification over exactly one set. This set is uniquely specified by a Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ formula that calculates a fixpoint. The syntax of Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ is defined as follows:

$$\varphi ::= (X, \chi, \varphi_{\mathrm{fp}}).\varphi \mid \forall \pi \in X.\varphi \mid \exists \pi \in X.\varphi \mid \psi \qquad \chi \in \{\curlyvee, \curlywedge\}$$

$$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \psi \,\mathcal{U}\, \psi \mid \bigcirc \psi \qquad a \in AP$$

If $\chi = \curlyvee$, $\varphi_{\mathrm{fp}}$ has to be a conjunction of formulas of the form

$$\forall \pi_1 \in X_1 \ldots \forall \pi_n \in X_n.\psi_{\mathit{step}} \rightarrow \pi_M \triangleright X$$

where $X_1, \ldots, X_n$ are previously quantified sets or $X$ and $1 \leqslant M \leqslant n$. The semantics of the restricted second-order quantification are:

$$\Pi, \Delta \models (X, \chi, \varphi_{\mathrm{fp}}).\varphi \qquad \text{iff there exists } A \in \mathit{sol}(\Pi, \Delta, (X, \chi, \varphi_{\mathrm{fp}}) : \Pi, \Delta[X \mapsto A] \models \varphi$$

The *sol* function returns the smallest sets satisfying $\varphi_{\mathrm{fp}}$:

$$\mathit{sol}(\Pi, \Delta, (X, \curlyvee, \varphi_{\mathrm{fp}})) := \{A \subseteq T \mid \Pi, \Delta[X \mapsto A] \models \varphi_{\mathrm{fp}} \wedge \forall A' \subsetneq A.\Pi, \Delta[X \mapsto A] \not\models \varphi_{\mathrm{fp}}\}$$

Note that the smallest does not refer to the sets with the least elements, but to the sets for which no subset satisfies the given condition.

## 3.5 Complexity Classes

A complexity class is a set of problems (or languages) that can be decided by the same kind of Turing machine. The fact that some Turing machine $M$ decides some language $L$ means in this context that $M$ accepts exactly all words in $L$. We use the usual definitions of complexity classes in this thesis. In particular, this includes P, EXP, PSPACE, EXPSPACE, $\Sigma_k^P$, $\Pi_k^P$, $\Sigma_k^{EXP}$ and $\Pi_k^{EXP}$.

**Definition 3.4** (P)

The complexity class P contains a problem $L$ if there is a polynomial $p$ and a deterministic Turing machine $M$ such that the following conditions hold:

- $M$ decides $L$.
- For an input of size $n$, $M$ terminates after at most $p(n)$ steps.

The complexity classes EXP as well as PSPACE are defined in a very similar way:

**Definition 3.5** (EXP)

The complexity class EXP contains a problem $L$ if there is a polynomial $p$ and a deterministic Turing machine $M$ such that the following conditions hold:

- $M$ decides $L$.
- For an input of size $n$, $M$ terminates after at most $\mathcal{O}(2^{p(n)})$ steps.

**Definition 3.6** (PSPACE)

The complexity class PSPACE contains a problem $L$ if there is a polynomial $p$ and a deterministic Turing machine $M$ such that the following conditions hold:

- $M$ decides $L$.
- For an input of size $n$, $M$ does not require more than $p(n)$ space.

The exponential counterpart of PSPACE is EXPSPACE and it is defined as follows:

**Definition 3.7** (EXPSPACE)

The complexity class EXPSPACE contains a problem $L$ if there is a polynomial $p$ and a deterministic Turing machine $M$ such that the following conditions hold:

- $M$ decides $L$.
- For an input of size $n$, $M$ does not require more than $\mathcal{O}(2^{p(n)})$ space.

There are several equivalent definitions for the polynomial hierarchy consisting of the complexity classes $\Sigma_k^p$ and $\Pi_k^p$. We will use the definition over alternating Turing machines (ATM) [AB09].

**Definition 3.8** ($\Sigma_k^p$)

Complexity class $\Sigma_k^p$ contains a problem $L$ if there is a polynomial $p$ and an alternating Turing machine $M$ such that the following conditions hold:

- $M$ decides $L$.
- For an input of size $n$, $M$ terminates after at most $p(n)$ steps.
- Every possible calculation of $M$ alternates at most $k-1$ times between universal and existential states.
- The initial state of $M$ is existential.

$\Pi_k^p$ is then defined as co-$\Sigma_k^p$. Similarly, there exists an exponential hierarchy:

**Definition 3.9** ($\Sigma_k^{EXP}$)

Complexity class $\Sigma_k^{EXP}$ contains a problem $L$ if there is a polynomial $p$ and an alternating Turing machine $M$ such that the following conditions hold:

- $M$ decides $L$.
- For an input of size $n$, $M$ terminates after at most $\mathcal{O}(2^{p(n)})$ steps.
- Every possible calculation of $M$ alternates at most $k-1$ times between universal and existential states.
- The initial state of $M$ is existential.

Again, we define $\Pi_k^{EXP}$ as co-$\Sigma_k^{EXP}$.

# 4 The Complexity of MC[Fixpoint]

In this chapter, we first give the formal problem statement. Then we show that the complexity of model checking Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ on tree-shaped structures is P-complete (Sect. 4.2) and EXP-complete on acyclic structures (Sect. 4.3).

## 4.1 Problem Statement

**Definition 4.1** (Hyper$^2$LTL Model Checking)
Given Kripke structure $\mathsf{K}$ and Hyper$^2$LTL formula $\varphi$, decide whether $\mathsf{K} \models \varphi$ holds.

This model checking problem was already shown to be undecidable [Beu+23]. The four model checking problems analyzed in this thesis impose further restrictions on $\mathsf{K}$ and $\varphi$:

- MC[Fixpoint, Tree] is the model checking problem where $\mathsf{K}$ is tree-shaped and $\varphi$ is a valid Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ formula
- MC[Fixpoint, Acyclic] is the model checking problem where $\mathsf{K}$ is acyclic and $\varphi$ is a Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ formula
- MC[Hyper$^2$LTL, Tree] is the model checking problem where $\mathsf{K}$ is tree-shaped and $\varphi$ is not further restricted
- MC[Hyper$^2$LTL, Acyclic] is the model checking problem where $\mathsf{K}$ is acyclic and $\varphi$ is not further restricted

If not specified otherwise, we assume in this thesis that only $\mathsf{K}$ is part of the input. This means that the complexities are given in the size of the model and not in the size of the formula.

## 4.2 P-completeness of MC[Fixpoint, Tree]

To prove that MC[Fixpoint, Tree] is P-complete, we will first show that it is contained in P and then we give a reduction in logarithmic space from the Horn-satisfiability problem to show that it is P-hard.

**Lemma 1.** *MC[Fixpoint, Tree] is contained in P.*

*Proof.* For every quantifier, we iterate over every possible instantiation and check if the inner formula holds for the current instantiation. We save instantiations of second-order quantifiers i.e. sets of traces by marking the corresponding leaves. We prove that this algorithm has a runtime that is polynomial in the size of the given structure by structural induction over Hyper$^2$LTL$_{fp}$ formulas.

$(X, \chi, \varphi_{fp}).\varphi$ Second-order quantifiers are restricted to sets that can be represented as a fixpoint. We can find this fixpoint in polynomial time because the set can only contain polynomial many traces such that we only have to do polynomially many fixpoint iterations. Each of these iterations can be done and checked in polynomial time by induction hypothesis.

$\forall \pi \in X.\varphi, \exists \pi \in X.\varphi$ First-order quantifiers can be checked in polynomial time because $\varphi$ can be checked in polynomial time and the number of possible instantiations of $\pi$ is polynomial in the size of the Kripke structure. This is because $X$ is fixed and can only contain traces from the Kripke structure, which is a tree.

$\psi$ $\psi$ is a Hyper-LTL formula and can be checked in polynomial time [BF18].

□

### 4.2.1 P-hardness of MC[Fixpoint, Tree]

To show that MC[Fixpoint, Tree] is P-hard we reduce the Horn satisfiability problem which is P-hard [Pap94], to MC[Fixpoint, Tree]. This means, that we built a tree $K$ and a Fixpoint Hyper$^2$LTL$_{fp}$ formula $\varphi$ from a given Horn-formula $h$ such that $K \models \varphi$ if and only if $h$ is true.

**Definition 4.2** (Horn-satisfiability)

Given a Boolean formula $h$ consisting of the conjunction of $n$ Horn clauses of the form $(\neg l_1 \vee \neg l_2 \vee l_3)$, where $l_1, l_2, l_3$ are from a set of numbered literals $\{x_1, \ldots, x_k\}$, $\top$ or $\bot$. The Horn-satisfiability problem is to find an assignment for the literals $x_1, \ldots, x_k$ such that $h$ is satisfied.

For simplicity, we add literals $x_{k+1}$ and $x_{k+2}$ as literals representing $\top$ and $\bot$ and restrict the valid assignments such that $x_{k+1}$ has to be assigned to true and $x_{k+2}$ has to be assigned to false.

Intuitively, we build a tree that has one branch for each literal that represents a positive assignment to this literal and one branch that represents a negative assignment to this literal. Additionally, there is one branch for every clause. On every branch, the number of the corresponding literal(s) is binary encoded using the atomic propositions

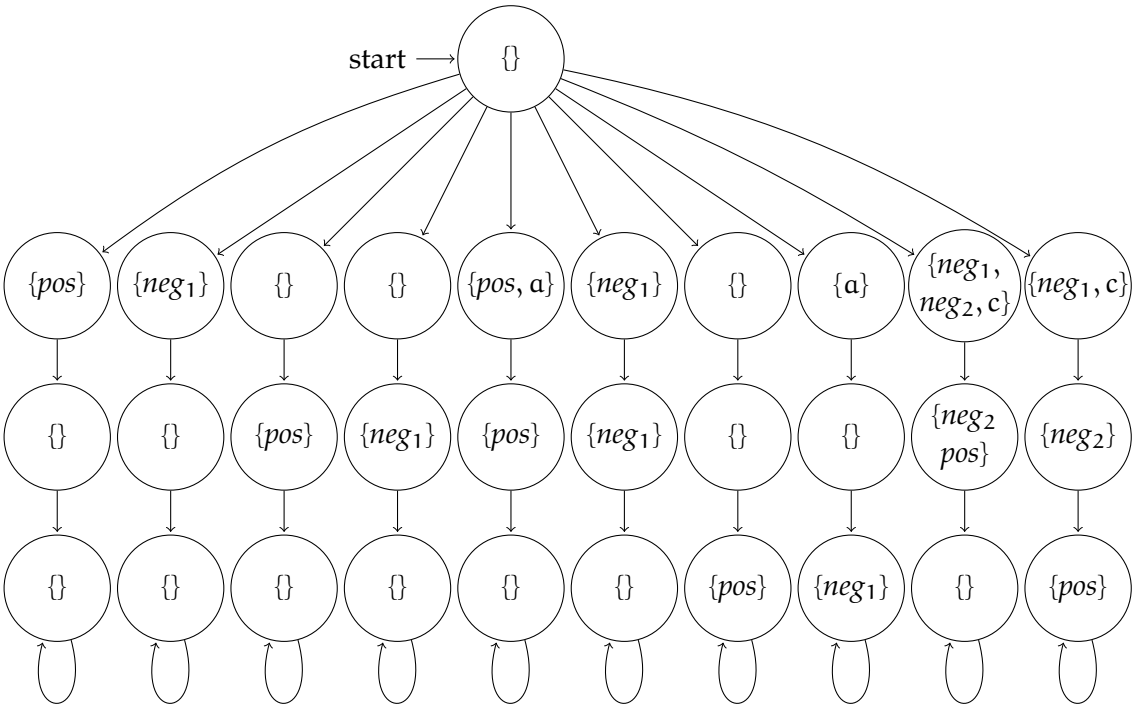



Figure 4.1: Kripke structure built for the Horn-formula $(\neg x_1 \vee \neg \top \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \bot)$. The two rightmost branches represent the clauses, the other branches represent positive resp. negative values for $x_1, x_2, \top, \bot$ (from left to right).

*pos*, $neg_1, neg_2$. The tree built for Horn-formula $(\neg x_1 \vee \neg \top \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \bot)$ can be seen in Fig. 4.1.

The Fixpoint Hyper$^2$LTL$_{fp}$ formula then constructs a set $A$, which represents the assignment. $A$ contains, for every literal, the positive trace if true is assigned to the literal and the negative trace if false is assigned to this literal. $A$ is constructed iteratively by adding an assignment trace to $A$ only if there is a clause that can not be satisfied if that literal would be assigned to the other value. To make sure that $x_{k+1}$ is assigned to true and $x_{k+2}$ is assigned to false, we mark the respective traces and add them to $A$.

We now give the formal reduction. To reduce the Horn satisfiability problem to MC[FIXPOINT, TREE] we build a Kripke structure $K = (S, s_0, \delta, L)$ whose graph is a rooted tree. The state-space $S$ consists of the initial state $s_0$ and $2 \cdot \lceil \log_2(k+3) \rceil$ states for each literal (including $\top, \bot$) and $\lceil \log_2(k+3) \rceil$ states for each clause in $h$. The states for a literal $x_i$ with $i \in [1, k]$ are called $s_{i,1}, s_{i,2}, \ldots$ and $s'_{i,1}, s'_{i,2}, \ldots$. The states for the $i$-th clause of $h$ are called $t_{i,1}, t_{i,2}, \ldots$.

The transition function $\delta$ connects $s_0$ to the states $s_{i,1}, s'_{i,1}$ for all $i \in [1,k]$ and $t_{j,1}$ for all $j \in [1,n]$. The states $s_{i,m}, s'_{i,m}, t_{i,m}$ are connected as branches with self-loops at the end. The function $\delta$ then looks as follows:

$$\delta(s_0) = \{s_{i,1} \mid i \in [1,k]\} \cup \{s'_{i,1} \mid i \in [1,k]\} \cup \{t_{j,1} \mid j \in [1,n]\}$$

$$\delta(s_{i,m}) = \begin{cases} \{s_{i,m+1}\} & \text{if } m < \lceil \log_2(k+3) \rceil \\ \{s_{i,m}\} & \text{else} \end{cases}$$

$$\delta(s'_{i,m}) = \begin{cases} \{s'_{i,m+1}\} & \text{if } m < \lceil \log_2(k+3) \rceil \\ \{s'_{i,m}\} & \text{else} \end{cases}$$

$$\delta(t_{i,m}) = \begin{cases} \{t_{i,m+1}\} & \text{if } m < \lceil \log_2(k+3) \rceil \\ \{t_{i,m}\} & \text{else} \end{cases}$$

The set of atomic propositions consists of $pos, neg_1, neg_2, c, a$. Only the states $t_{1,1}, \ldots, t_{n,1}$ are labeled with $c$ and only the states $s_{k+1,1}$ and $s'_{k+2,1}$ are labeled with $a$ to mark the branches representing $\top$ and $\bot$.

$pos, neg_1, neg_2$ only occur on states $s_{i,m}, s'_{i,m}$ and $t_{i,m}$. The traces containing states $s_{i,m}$ are labeled with $pos$ such that they have $i$ binary encoded on them. Similarly, the traces consisting of states $s'_{i,m}$ are labeled with $neg_1$ such that they have $i$ binary encoded on them. A trace consisting of the states $t_{i,m}$ representing the $i$-th clause $(\neg x_a, \neg x_b, x_c)$ has $a$ binary encoded with $neg_1$, $b$ binary encoded with $neg_2$ and $c$ binary encoded with $pos$ on it. An example of the finished structure can be seen in Fig. 4.1.

This results in a Kripke structure in the form of a tree. For each literal $x_i$, we have two branches representing a positive or negative assignment to $x_i$. The branches have the number $i$ binary encoded on the states of the branch using the propositions $pos$ and $neg_1$. Similarly, there is a branch for each clause in $h$. These branches have the number of the one positive and two negative literals encoded on their states using the propositions $pos, neg_1, neg_2$. To distinguish branches representing clauses from the branches representing literals, they are labeled with $c$. Additionally, the branches representing a positive assignment for $\top$ and a negative assignment for $\bot$ are marked with $a$ to hint that there is no valid assignment without these two traces.
Note that the size of the structure is polynomial in the size of $h$. The formula $\varphi$, for which we check whether or not it is satisfied by $K$, looks like this:

$$(A, \Upsilon, \varphi_{fp} \wedge \varphi'_{fp}).\forall \pi \in A.\forall \pi' \in A.\neg\Box(pos_\pi \leftrightarrow neg_{1,\pi'})$$

Where $\varphi_{fp}$ is

$$\forall\pi \in \mathfrak{G}.\forall\alpha \in A.\forall\alpha' \in A.\forall\beta \in \mathfrak{G}. \bigcirc c_\pi \wedge \neg \bigcirc c_\beta \wedge \pi \neq_{AP} \alpha \neq_{AP} \alpha' \neq_{AP} \beta \quad (1)$$

$$\wedge(\Box(pos_\alpha \leftrightarrow neg_{1,\pi}) \wedge \Box(pos_{\alpha'} \leftrightarrow neg_{2,\pi}) \wedge \Box(pos_\beta \leftrightarrow pos_\pi) \quad (2)$$

$$\vee \Box(pos_\alpha \leftrightarrow neg_{1,\pi}) \wedge \Box(neg_{1,\alpha'} \leftrightarrow pos_\pi) \wedge \Box(neg_{1,\beta} \leftrightarrow neg_{2,\pi}) \quad (3)$$

$$\vee \Box(pos_\alpha \leftrightarrow neg_{2,\pi}) \wedge \Box(neg_{1,\alpha'} \leftrightarrow pos_\pi) \wedge \Box(neg_{1,\beta} \leftrightarrow neg_{1,\pi})) \quad (4)$$

$$\rightarrow\beta \triangleright A \quad (5)$$

and $\varphi'_{fp}$ is

$$\forall\pi \in \mathfrak{G}. \bigcirc a_\pi \rightarrow \pi \triangleright A$$

$\varphi$ quantifies over the smallest set $A$, which will represent the current assignment for the Horn satisfiability problem and checks that there is no literal for which the positive as well as the negative trace is in $A$. $\varphi_{fp}$ makes sure that a trace $\beta$ is only added to $A$ (line 5) if there exist assignment traces $\alpha, \alpha'$ which are already in $A$ and clause trace $\pi$, which are all different (line 1) and satisfy the condition in lines 2 to 4. The condition in lines 2 to 4 is satisfied exactly if $\alpha$ and $\alpha'$ represent an assignment to two of the literals in the clause represented by $\pi$ such that this clause can only be satisfied if the third literal is represented by $\beta$ and has to be assigned as $\beta$ represents.

This is the case only if $\alpha$ and $\alpha'$ represent a positive assignment to both negative literals and $\beta$ represents a positive assignment to the positive literal (l. 2) or $\alpha$ represents a positive assignment to one of the negative literals and $\alpha'$ represents a negative assignment to the positive literal and $\beta$ represents a negative assignment to the other negative literal (l. 3-4).

Next, we show that this reduction is correct.

**Lemma 2.** $K \models \varphi$ *if and only if the given Horn formula is satisfiable.*

*Proof.* We start by proving that the Horn formula is satisfiable if $K \models \varphi$. If the answer to the model checking problem is positive, then we can construct a satisfying assignment from the set $A$. If the trace that represents the positive form of a literal is in $A$, then we assign this literal to true, if the trace representing a negative form of a literal is in $A$, then we assign this literal to false. If there is neither the positive nor the negative trace for a literal in $A$, we assign this literal to false and if the positive as well as the negative trace is contained in $A$, then the answer to the model checking problem is not positive.

We prove that this assignment is satisfying by using proof by contradiction. If the assignment would not be satisfying, then there is a clause $c$, which is not satisfied. We distinguish two cases:

- If there are two or three literals in $c$ for which neither the trace representing positive assignment nor the trace representing negative assignment is contained in $A$, then there are at least two literals in $c$ assigned to false, which satisfies $c$.
- If there is one or no literal in $c$ for which neither the positive nor the negative trace is contained in $A$, then $A$ does contain traces $\alpha$ and $\alpha'$ for two of the literals in $c$, which do not satisfy $c$. $\varphi_{fp}$ then enforces that the trace satisfying the remaining literal in $c$ is an element of $A$. This leads to an assignment that satisfies $c$ if only one of the traces for the literal is contained in $A$ or it leads to a resulting set $A$ which contains the positive as well as the negative trace for the remaining literal, which would then contradict the assumption that the answer to the model checking problem is positive.

It remains to show that $K \not\models \varphi$ implies that the given Horn formula is unsatisfiable. We first prove that if a trace representing the assignment of one literal to true is in $A$, then there is no assignment satisfying the Horn formula, where this literal is assigned to false and vice versa. More intuitively, we only add assignment traces whose assignment must be necessarily correct. We do natural induction over the number of fixpoint iterations.

**Base Case:** $\varphi'_{fp}$ enforces that at least the traces assigning $\top$ to true and $\bot$ to false are part of $A$. Per definition of the Horn-satisfiability problem, there is no valid assignment that assigns $\top$ to false or $\bot$ to true. Because $A$ is empty in the beginning $\varphi_{fp}$ does not add any traces in the first fixpoint iteration.

**Induction Case:** For every trace $\beta$ that is in $A$, there have to exist three traces: $\pi \in G$ representing a horn clause $c$ and $\alpha, \alpha' \in A$ representing two assignments to literals. All four traces satisfy $\varphi_{fp}$. If this would not be the case, then $\beta$ does not have to be included in $A$ and $A$ would not be the smallest set satisfying $\varphi_{fp} \wedge \varphi'_{fp}$. The traces satisfy $\varphi_{fp}$ exactly iff $\alpha$ and $\alpha'$ represent an assignment for two of the literals in $c$ in such a way, that $c$ can only be satisfied if the third literal is assigned according to $\beta$. By the induction hypothesis, we know that there does not exist a valid assignment satisfying the Horn formula, which conflicts with $\alpha$ or $\alpha'$. That means that there is also no satisfying assignment conflicting $\beta$.

The answer to the model checking problem is only negative if there exist two traces in $A$ which represent the assignment of the same literal to true and to false. We can now conclude that in this case, there is no satisfying assignment to the Horn formula which does not assign true as well as false to the same literal, which would result in an invalid assignment. □

**Lemma 3.** *The given reduction needs space that is logarithmic in the size of* $h$.

*Proof.* To show that this reduction only needs space which is logarithmic to the size of $h$ let $k$ be the highest number of a literal in the Horn formula and let $n$ be the number of

clauses in the Horn formula. The Kripke structure has $n + 4 + 2k$ many branches with a depth of $\lceil \log_2(k+3) \rceil$. These branches can be built sequentially by only remembering in which branch we are and at what depth. This needs only $\lceil \log_2(n + 4 + 2k) \rceil$ or $\lceil \log_2(\lceil \log_2(k + 3) \rceil) \rceil$ space respectively. With the first counter we can compute the corresponding literal or find the corresponding clause in the input and with the second counter we decide if the binary representation of the relevant literal(s) has a 1 at this position and if the current node has to be a leaf. □

**Theorem 4.** *MC[FIXPOINT, TREE] is P-complete*

*Proof.* By Lem. 2 and Lem. 3 follows that MC[FIXPOINT, TREE] is P-hard. By this and Lem. 1 follows that MC[FIXPOINT, TREE] is P-complete. □

## 4.3 EXP-completeness of MC[Fixpoint, Acyclic]

**Lemma 5.** *MC[Fixpoint, Acyclic] is contained in EXP*

*Proof.* We reduce MC[Fixpoint, Acyclic] to MC[Fixpoint, Tree]. An acyclic model can not have more than $2^{p(n)}$ traces, where $p(n)$ is some polynomial in the number of states of the model. We can iterate over all traces in a manner similar to depth-first search.

Let $K$ be the given acyclic Kripke structure with $n$ states and $t \leqslant 2^{p(n)}$ traces and let $\varphi$ be the given Fixpoint Hyper$^2$LTL$_{fp}$ formula. We construct a tree-shaped Kripke structure $T$ which contains all traces in $K$ by iterating over all traces in $K$ and adding for each trace in $K$ a new branch with equally labeled states to $T$. This can be done in exponential time because $t$ is exponential in $n$ and adding a trace to $T$ does not take longer than the length of the trace which is bound by $n$. If $|T|$ is the number of states of $T$, then $|T| \leqslant n \cdot t \leqslant n \cdot 2^{p(n)}$ holds.

Because the sets of traces in $K$ and $T$ are equal $K \vDash \varphi$ holds exactly if $T \vDash \varphi$ holds.
→ Lem. 1, p. 16 Therefore we can use the algorithm from Lem. 1 whose runtime is less or equal to some polynomial $p'$ in the size of the given structure. If we run this algorithm on $T$ its runtime can be expressed as $p'(|T|) \leqslant p'(2^{p(n)})$ which implies the existence of a polynomial $p''$ such that $p'(2^{p(n)}) \leqslant 2^{p''(n)}$.

Because the construction of $T$ can be done in exponential time and the runtime of the MC[Fixpoint, Tree] algorithm is exponential on $T$, MC[Fixpoint, Acyclic] can be solved in exponential time. □

### 4.3.1 EXP-hardness of MC[Fixpoint, Acyclic]

To prove that MC[Fixpoint, Acyclic] is EXP-hard, we give a polynomial-time many-one reduction from Succinct Circuit Value.

We use very similar definitions as [Pap94].

**Definition 4.3** (Boolean Circuit)

A Boolean circuit is a directed acyclic Graph $C = (V, E)$. The nodes are called gates and every gate $u$ is annotated with a sort $sort(u)$ which is TRUE, FALSE, AND, OR, NOT or INPUT. If a gate is of sort TRUE, FALSE or INPUT, it has an indegree of 0. Gates of sort NOT have indegree 1 and gates of sort AND or OR have an indegree of 2. The gates with outdegree 0 are called output gates. Input gates as well as output gates are numbered.

The semantics of Boolean circuits for some input are defined by a function $\mathcal{T}$ that maps gates to Boolean values. An input to a circuit is a bitstring that assigns one of its bits to each input gate. The value of $\mathcal{T}(u)$ for $u \in V$ is defined inductively.

- If $u$ is an input gate, then $\mathcal{T}(u)$ is the value of the corresponding input bit.
- If $u$ is of sort TRUE or FALSE, then $\mathcal{T}(u) = \top$ resp. $\mathcal{T}(u) = \bot$.

- If $u$ is of sort NOT, then there exists exactly one gate $v$ such that $(v, u) \in E$. $\mathcal{T}(u)$ is then $\neg\mathcal{T}(v)$.
- If $u$ is of sort OR, then there are exactly two gates $v, v'$ such that $(v, u), (v', u) \in E$. $\mathcal{T}(u)$ is defined as $\mathcal{T}(v) \vee \mathcal{T}(v')$.
- If $u$ is of sort AND, then there are exactly two gates $v, v'$ such that $(v, u), (v', u) \in E$. $\mathcal{T}(u)$ is defindes as $\mathcal{T}(v) \wedge \mathcal{T}(v')$.

We say an edge $(u, v)$ carries a positive value if $\mathcal{T}(u) = \top$ and we say that $(u, v)$ carries a negative value otherwise. The output of a Boolean circuit is the bitstring of the values assigned to the output gates by $\mathcal{T}$.

**Definition 4.4** (Succinct Representation)

We say that a Boolean circuit $C_S$ *succinctly represents* a Boolean circuit $C$ if and only if $C_S$ represents $C$ as follows: The gates of $C$ are numbered from 1 to $m$. The neighbors of a gate $u$ are all gates $v$ such that $(u, v) \in E$ or $(v, u) \in E$. The neighbors of gate $u$ are also numbered such that the first up to two neighbors are the predecessors of $u$ and the rest are its successors.

Let $k = \lceil \log_2(m) \rceil$ which implies $2^k \geqslant m$. $C_S$ has $2k$ input gates and $k+3$ output gates.

Let the input to $C_S$ be of the form $i * j$[1] where $i$ and $j$ both consist of $k$ bits. Let $q * r$ be the corresponding output where $q$ consists of $k$ bits and $r$ consists of three bits. Let $u \in V$ be the gate with number $i$ and let $v \in V$ be the gate with number $q$. The encoding is such that the $j$-th neighbor of $u$ is gate $v$ and $r$ encodes the kind of $u$. If $u$ has no $j$-th neighbor, then the neighbor is some fictitious gate with the number 0.

$sort(u)$ for a gate $u$ is encoded in $r$ as follows: $TRUE \to 001, FALSE \to 010, AND \to 011, OR \to 100, NOT \to 101$

**Definition 4.5** (Succinct Circuit Value)

Given a Boolean circuit $C_S$ which succinctly represents a Boolean circuit $C$, decide whether $\mathcal{T}$ is true for the only output gate of $C$.

We denote the set of all gates in $C_S$ and $C$ as $V(C_S)$ resp. $V(C)$ and we denote the set of all edges in $C_S$ and $C$ as $E(C_S)$ resp. $E(C)$. Additionally, we denote the number of input gates of $C_S$ as $|e|$ and the number of gates it has as $n$.

Note that every input-output pair of $C_S$ represents an edge in $C$.

To reduce Succinct Circuit Value to MC[FIXPOINT, ACYCLIC] we build a Kripke structure $K$ from the given circuit $C_S$ and a Fixpoint Hyper$^2$LTL$_{fp}$ formula $\varphi$ such that $K \models \varphi$ if and only if the Succinct Circuit Value Problem is true for $C_S$. The reduction consists of

[1] where $*$ is the concatenation of two bitstrings

three phases. The first two phases collect the output of $C_S$ for every input and the last phase solves the Circuit Value problem on $C$.

Intuitively, in phase A some set $X$ is built such that $X$ contains traces that encode the values for all edges in $C_S$ for all possible inputs. Therefore all traces have a bitstring representing the input of $C_S$ and two gate IDs on them; one for the gate they are coming from and one for the gate they are going to. Additionally, the type of gate the edge is going to is marked on the trace and whether the trace encodes a positive or negative value on the edge. For every edge in $C_S$ and every input $e$, $X$ contains the positive trace if the edge carries a positive value under input $e$ and the negative trace otherwise.

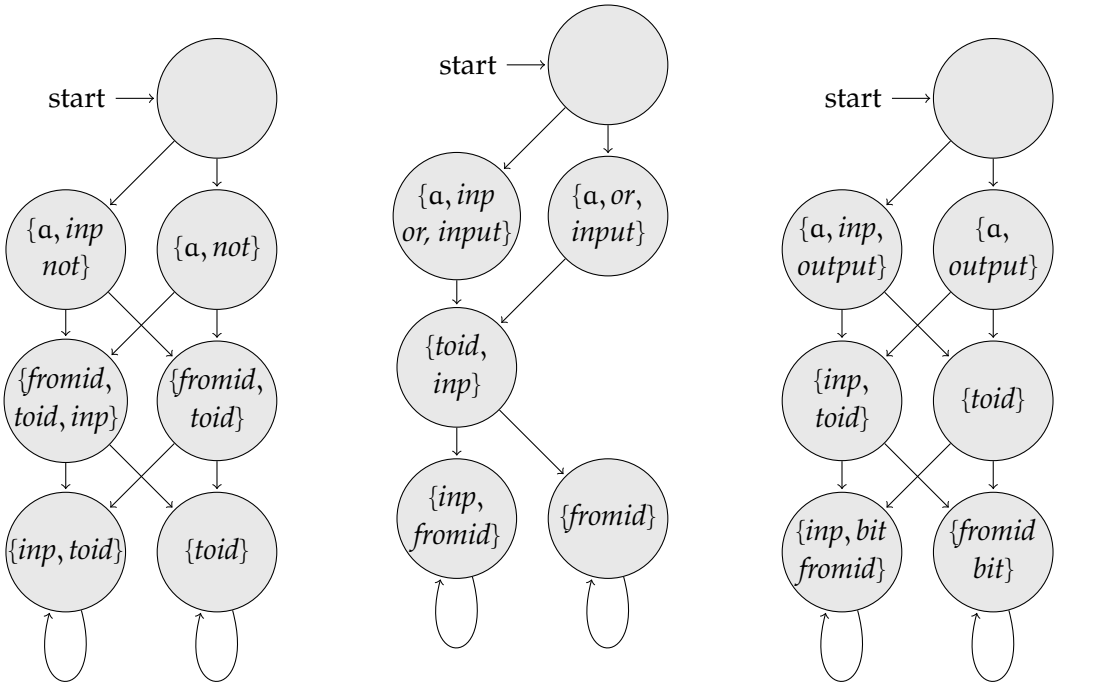
In phase B, we build a set $Y$ that contains for each input one trace encoding the input and the corresponding output of $C_S$. For this, every trace in phase B has two bitstrings on them; one representing an input to $C_S$ and one representing the corresponding output of $C_S$. We build the set $Y$ from set $Y'$. $Y'$ contains traces where the encoded output ends before the trace has reached its self-loop. We call these traces with a partial output *incomplete*. The fixpoint iteration for $Y'$ adds, with every iteration traces that have one bit more output than the previous iteration. In the last step of the iteration, the traces are long enough to encode a complete input-output pair. We can compute set $Y$ by taking all traces from $Y'$ that are not incomplete.

In phase C, all information about the Boolean circuit $C$ is available in set $Y$. In this phase, all traces are traces from $Y$ which additionally have a mark whether they represent a positive or negative value. We construct set $Z$ such that for each trace in $Y$ the positive trace is included in $Z$ if and only if the represented edge is positive in $C$. If the represented edge is negative in $C$, then we add the negative trace to $Z$. As the last step, we check whether $Z$ contains a trace representing the output edge. The answer to the Succinct Circuit Value Problem is true if and only if this trace is in $Z$.

We now describe the reduction formally. Given a circuit $C_S$, we build a Kripke structure $K$ from $C_S$. The formula $\varphi$ is independent of $C_S$.

### Phase A

In phase A we construct a Kripke structure representing all edges in $C_S$. For this, we first modify $C_S$. First, we eliminate all binary gates that have two inputs coming from the same gate. It follows, that there are no two edges $(u, v), (s, t) \in E(C_S)$ with $u = s$ and $v = t$. Then, we replace the TRUE and FALSE gates in $C_S$ by circuits representing $p \wedge \neg p$ resp. $p \vee \neg p$, where $p$ is some input gate. We reorder the input gates specifying the neighbor number, such that they are reversed. This shortens the construction in phase C. Further, we add to every gate that produces an output of $C_S$ an output edge going to a special gate of sort OUTPUT which we call *output gate*. Additionally, we give every gate $u$ in $C_S$ a unique ID greater than zero denoted by $\mathrm{ID}(u)$. The IDs are distributed such that for every $(u, v) \in E(C_S)$ holds $\mathrm{ID}(u) < \mathrm{ID}(v)$. Finally, we calculate $l \in \mathbb{N}$ such that traces of length $l$ are long enough to encode one ID of a gate in $C_S$ as well as inputs and outputs of $C_S$ e.g. $l = \max(\lceil \log_2(n) \rceil, |e|, o)$ if $C_S$ has $o$ output gates.




(a) The positive substructure for an edge between gates $u$ and $v$ with $\mathrm{ID}(u) = 2, \mathrm{ID}(v) = 3$ and $v$ being a NOT gate. Every state is additionally labeled with *pos*.

(b) The positive substructure for an edge between gates $u$ and $v$ with $\mathrm{ID}(u) = 1, \mathrm{ID}(v) = 2$ and $v$ being an OR gate. Gate $u$ takes the second input bit. Every state is additionally labeled with *pos*.

(c) The negative substructure for an edge between gates $u$ and $v$ with $\mathrm{ID}(u) = 1, \mathrm{ID}(v) = 2$. Gate $v$ is the third output bit. Every state is additionally labeled with *neg*.

Figure 4.2: Substructures for phase A with $l = |e| = 3$.

In the case where $|e| < l$ we add $l - |e|$ *dummy input gates* to $C_S$ which are not connected to any other gate. Similarly, if $o < l$ we add $l - o$ *dummy output gates* to $C_S$ which always output false. $\mathrm{ID}(u)$ is undefined if $u$ is a dummy input gate or dummy output gate. We call every input where the inputs to the dummy input gates are false *valid*.

**Def.** valid input

For every edge, we build two substructures connected to the initial state in $K$. These substructures consist of $l$ layers with two states per layer. The purpose of the structures is to represent a positive resp. negative value on the edge.

Let $a_{u,v,i,j,pos}$ be the $j$-th state in the $i$-th layer of one of the substructures representing the edge $(u, v) \in E(C_S)$ and let $a_{u,v,i,j,neg}$ be the same state in the other substructure for $(u, v)$. Note that $i \in \{1, \ldots, l\}$ and $j \in \{1, 2\}$. In every substructure the two states of one layer are fully connected to the states of the next layer and the states of the last layer have self-loops attached. There are no other transitions in this substructure (see Fig. 4.2a).

All states $a_{u,v,i,j,pos}$ are labeled with *pos*. All states $a_{u,v,i,j,neg}$ are labeled with *neg*. Additionally states $a_{u,v,1,j,pos}$ and $a_{u,v,1,j,neg}$ are labeled with atomic proposition $a$ to represent that they are part of phase A. We label the states $a_{u,v,i,j,pos}$ and $a_{u,v,i,j,neg}$ with the atomic proposition *fromid* if and only if the $i$-th bit of $\mathrm{ID}(u)$ is 1 and equally with atomic proposition *toid* for $\mathrm{ID}(v)$. This encodes $u$ and $v$ on each substructure.

We use atomic propositions *output*, *and*, *or*, *not* to encode $sort(v)$. States $a_{u,v,1,j,pos}$ and $a_{u,v,1,j,neg}$ are labeled with one of these propositions.

States $a_{u,v,i,2,pos}$ and $a_{u,v,i,2,neg}$ with $i \leqslant |e|$ are labeled with *inp*. An example of how a completely labeled structure looks like can be seen in Fig. 4.2a.

If $u$ is the input gate which takes the $m$-th input, then holds that states $a_{u,v,1,j,pos}$ and $a_{u,v,1,j,neg}$ are labeled with *input*. Additionally, there are no states $a_{u,v,m,1,pos}$ and $a_{u,v,m,2,neg}$.

For the edges going to the $m$-th output gate the states $a_{u,v,m,j,pos}$ and $a_{u,v,m,j,pos}$ are labeled with atomic proposition *bit*. Example substructures for edges leaving input gates and entering output gates can be seen in Fig. 4.2b and Fig. 4.2c.

For every dummy output gate responsible for the $m$-th output of $C_S$ we build one substructure. This substructure looks exactly like a negative substructure for a normal output gate but there are no gate IDs encoded on this substructure. These substructures are additionally labeled with atomic proposition *dummy* in the first states.

To summarize which traces are now part of $K$ and labeled with $a$ somewhere: For each edge $(u, v) \in E(C_S)$ and each valid input $e$ we have one trace labeled with *pos* and one trace labeled with *neg*. Both of these traces have the IDs of $u$ and $v$ as well as $e$ and the type of gate $v$ encoded on them. Positive traces representing an edge leaving the $m$-th input gate only exist if the $m$-th bit of $e$ is a 1. The corresponding negative traces only exist if the $m$-th bit of $e$ is a 0. For each edge going to the $m$-th output gate the traces representing that edge are labeled with *bit* in their $m + 1$-st state. There are for every valid input $e$ and every dummy output gate - responsible for the $m$-th output - traces encoding $e$, labeled with *bit* at the $m + 1$-st step and labeled with *neg*. These are the only traces labeled with atomic proposition *dummy*.

Set $X$ should contain a trace representing edge $(u, v)$ labeled with *pos* and input $e$ if and only if the value of edge $(u, v)$ is positive for input $e$. Similarly, the corresponding *neg* trace should be contained in $X$ if and only if the value of edge $(u, v)$ is negative for input $e$. The fixpoint formula describing $X$ has different parts for each sort of gate, such that $\varphi$ looks as follows:

$$\varphi = (X, \Upsilon, \varphi_{input} \wedge \varphi_{and} \wedge \varphi_{or} \wedge \varphi_{not}).\varphi_B$$

Because AND and OR gates behave similarly to each other, $\varphi_{and}$ also adds negative traces for OR gates and $\varphi_{or}$ also adds negative traces for AND gates. We start by describing $\varphi_{input}$:

$$\varphi_{input} = \forall \pi \in \mathfrak{G}. \Diamond a_\pi \wedge \Diamond input_\pi \rightarrow \pi \triangleright X$$

$\varphi_{input}$ adds all traces representing edges leaving output gates to X.

$$\begin{aligned}\varphi_{and} =&(\forall\pi_1 \in \mathfrak{G}.\forall\pi_2,\pi_3 \in X.\Diamond(\pi_2 \neq_{AP} \pi_3)\\ &\wedge \Diamond a_{\pi_1} \wedge \Diamond a_{\pi_2} \wedge \Diamond a_{\pi_3}\\ &\wedge \Box((inp_{\pi_1} \leftrightarrow inp_{\pi_2}) \wedge (inp_{\pi_2} \leftrightarrow inp_{\pi_3}))\\ &\wedge \Box((toid_{\pi_2} \leftrightarrow toid_{\pi_3}) \wedge (toid_{\pi_3} \leftrightarrow fromid_{\pi_1}))\\ &\wedge ((\Diamond and_{\pi_2} \wedge \Diamond pos_{\pi_1} \wedge \Diamond pos_{\pi_2} \wedge \Diamond pos_{\pi_3})\\ &\quad \vee (\Diamond or_{\pi_2} \wedge \Diamond neg_{\pi_1} \wedge \Diamond neg_{\pi_2} \wedge \Diamond neg_{\pi_3}))\\ &\rightarrow \pi_1 \triangleright X)\end{aligned}$$

We quantify over three traces: $\pi_2$ and $\pi_3$ are two different traces already in X and $\pi_1$ is the trace we add if the step formula is satisfied. To make sure that we are only talking about traces relevant in phase A, we check whether all three traces are labeled with $a$. Additionally, all three traces should encode the same input for $C_S$ and $\pi_2$ and $\pi_3$ should point to the gate where $\pi_1$ is coming from. We add $\pi_1$ to set X if, additionally, all three traces represent positive values on the edges and the gate $\pi_2$ is going to is an AND gate. Alternatively, we add $\pi_1$ to X if all three traces represent negative values on their edges and $\pi_2$ is going to an OR gate.

$$\begin{aligned}\varphi_{or} =&(\forall\pi_1 \in \mathfrak{G}.\forall\pi_2 \in X.\\ &\Diamond a_{\pi_1} \wedge \Diamond a_{\pi_2}\\ &\wedge \Box(inp_{\pi_1} \leftrightarrow inp_{\pi_2})\\ &\wedge \Box(toid_{\pi_2} \leftrightarrow fromid_{\pi_1})\\ &\wedge ((\Diamond or_{\pi_2} \wedge \Diamond pos_{\pi_1} \wedge \Diamond pos_{\pi_2})\\ &\quad \vee (\Diamond and_{\pi_2} \wedge \Diamond neg_{\pi_1} \wedge \Diamond neg_{\pi_2}))\\ &\rightarrow \pi_1 \triangleright X)\end{aligned}$$

$\varphi_{or}$ adds positive traces for outgoing edges for OR gates with at least one positive input and negative edges for AND gates with at least one negative input. First, we check again whether both traces are from the substructures from phase A, have the same input encoded and whether $\pi_2$ is an input to the gate for which $\pi_1$ is an output. Then we add $\pi_1$ to X if it represents a positive output to an OR gate with one positive input or if $\pi_1$ represents a negative output for an AND gate with one negative input.

Similarly, for NOT gates we add $\pi_1$ if it represents the opposite value of $\pi_2$:

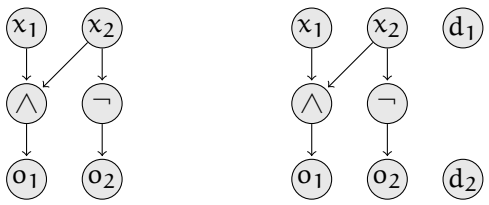



Figure 4.3: The circuit considered in Example 4.3.1

$$
\begin{aligned}
\varphi_{not} = (&\forall \pi_1 \in \mathfrak{G}. \forall \pi_2 \in X. \\
&\Diamond a_{\pi_1} \wedge \Diamond a_{\pi_2} \\
&\wedge \Box(inp_{\pi_1} \leftrightarrow inp_{\pi_2}) \\
&\wedge \Box(toid_{\pi_2} \leftrightarrow fromid_{\pi_1}) \\
&\wedge ((\Diamond not_{\pi_2} \wedge \Diamond pos_{\pi_1} \wedge \Diamond neg_{\pi_2}) \\
&\quad \vee (\Diamond not_{\pi_2} \wedge \Diamond neg_{\pi_1} \wedge \Diamond pos_{\pi_2})) \\
&\rightarrow \pi_1 \triangleright X)
\end{aligned}
$$

**Example 4.3.1.** Consider the case where $C_S$ is the circuit of Fig. 4.3 on the left. A valid distribution of IDs is from the top-left to the bottom-right with IDs one to six. Because there are six states we have to choose $l = 3$. The resulting circuit, after adding dummy gates $d_1$ and $d_2$ is depicted in Fig. 4.3 on the right. We call the states in this example by the symbol they are labeled with such that we have the states $x_1, x_2, \wedge, \neg, o_1, o_2, d_1$ and $d_2$ and $\mathrm{ID}(x_1) = 1, \mathrm{ID}(x_2) = 2, \mathrm{ID}(\wedge) = 3, \mathrm{ID}(\neg) = 4, \mathrm{ID}(o_1) = 5, \mathrm{ID}(o_2) = 6$.

$C_S$ has three input gates, three output gates and four valid inputs. The construction of $K$ and $\varphi_{input}$ enforces that $X$ contains the following traces:

- $t_1, t_2, t_3$: Three negative traces encoding input 000 and representing edges $(x_1, \wedge)$, $(x_2, \wedge)$ and $(x_2, \neg)$.
- $t_4$: One negative trace encoding input 010 and representing edge $(x_1, \wedge)$.
- $t_5, t_6$: Two positive traces encoding input 010 and representing edges $(x_2, \wedge)$ and $(x_2, \neg)$.
- $t_7$: One positive trace encoding input 100 and representing edge $(x_1, \wedge)$.
- $t_8, t_9$: Two negative traces encoding input 100 and representing edges $(x_2, \wedge)$ and $(x_2, \neg)$.

- $t_{10}, t_{11}, t_{12}$: Three positive traces encoding input 110 and representing edges $(x_1, \wedge), (x_2, \wedge)$ and $(x_2, \neg)$.

Once $t_1 - t_{12}$ are contained in X, $\varphi_{and}$ enforces that more traces are added to X:

- $t_{13}$: One positive trace encoding input 110 and edge $(\wedge, o_1)$ by instantiating $\pi_2$ with $t_{10}$ and $\pi_3$ with $t_{11}$.

Similarly, $\varphi_{or}$ is only satisfied if the following traces are also contained in X:

- $t_{14}$: One negative trace encoding input 000 and edge $(\wedge, o_1)$ by instantiating $\pi_2$ with $t_1$ or $t_2$.
- $t_{15}$: One negative trace encoding input 010 and edge $(\wedge, o_1)$ by instantiating $\pi_2$ with $t_4$.
- $t_{16}$: One negative trace encoding input 100 and edge $(\wedge, o_1)$ by instantiating $\pi_2$ with $t_8$.

Lastly, $\varphi_{not}$ requires the following traces in X to be satisfied:

- $t_{17}$: One positive trace encoding input 000 and edge $(\neg, o_2)$ by instantiating $\pi_2$ with $t_3$.
- $t_{18}$: One negative trace encoding input 010 and edge $(\neg, o_2)$ by instantiating $\pi_2$ with $t_6$.
- $t_{19}$: One positive trace encoding input 100 and edge $(\neg, o_2)$ by instantiating $\pi_2$ with $t_9$.
- $t_{20}$: One negative trace encoding input 110 and edge $(\neg, o_2)$ by instantiating $\pi_2$ with $t_{12}$.

By the construction of the substructures, the traces $t_{13}, t_{14}, t_{15}$ and $t_{16}$ are labeled with *bit* in the second step and $t_{17}, t_{18}, t_{19}$ and $t_{20}$ are labeled with *bit* in the third step. $t_{13}$ to $t_{16}$ represent the value of the first output bit for every input and $t_{17}$ to $t_{20}$ represent the value of the second output bit for every input. Because the input and output of $C_S$ is entirely represented by these traces and the traces labeled with *dummy*, they are the only traces relevant for phase B. △

**Lemma 6.** *The structure built in phase A can be built in time polynomial in the size of* $C_S$.

*Proof.* Because $C_S$ is explicitly given as input we can do all the preprocessing of $C_S$ in polynomial time in the size of the input. The IDs can be distributed by calculating a topological order on the gates of $C_S$ and we add at most $l$ dummy gates. The substructures can be built by iterating over every edge and building both relevant substructures because the form of the substructure only depends on the two gates the edge connects. Additionally, $l$ and the number of edges in $C_S$ are polynomial in the size of the input such that there are only polynomially many states and transitions in the structure. □

**Lemma 7.** *For all gates* $u, v \in V(C_S)$*, all inputs* $e$ *to* $C_S$ *and all atomic propositions* $d \in \{pos, neg\}$ *holds that a trace* $\pi$ *encoding* $ID(u), ID(v), e$ *and* $d$ *is an element of* $X$ *if and only if*

1. $(u, v) \in E(C_S)$
2. $e$ *is a valid input*
3. *for input* $e$ *this edge carries a positive value if* $d = pos$ *and a negative value if* $d = neg$

*Proof.* All formulas adding traces to $X$ only add traces that are labeled with $a$ and therefore come from the substructures described in phase A. We say that a trace $\pi$ represents edge $(u, v)$ if $\pi$ has $ID(u)$ encoded with *fromid* and $ID(v)$ encoded with *toid*.

Because these substructures only contain traces encoding edges of $C_S$ there can not be a trace representing an edge that does not exist in $C_S$.

Additionally, only the first $|e|$ layers are labeled with *inp* such that $e$ can not have positive inputs for dummy input gates. Therefore, every encoded input is valid.

Further, we observe that $sort(u)$ can be found as follows: Let $\pi$ represent edge $(u, v) \in E(C_S)$ and $\pi_2$ represent edge $(w, u) \in E(C_S)$. $\pi_2$ is labeled with the atomic proposition representing $sort(u)$.

We do complete induction over $ID(u)$ to show that $\pi$ is contained in $X$ if and only if $\pi$ represents the correct value. We distinguish between the different types of $u$.

**$u$ is an input gate:** By construction of the substructures only traces where $u$ is an input gate are labeled with *input*. We show that adding all traces labeled with *input* to $X$ adds $\pi$ to $X$ if and only if $\pi$ represents the correct value for the edge $(u, v)$. Let $m \in \mathbb{N}$ such that $u$ takes the $m$-th input bit of $C_S$. If $d$ is positive, then $\pi$ should be added if and only if the $m$-th bit of $e$ is 1. This is the case because in the only substructure containing traces that encode $u, v$ and *pos* the state $a_{u,v,m,1,pos}$ is missing and $\pi$ has to visit $a_{u,v,m,2,pos}$ which represents 1 as the $m$-th bit of $e$. The situation is analogous if $d$ is negative.

**$u$ is an AND gate:** If $u$ is an AND gate, then there are gates $s, s' \in V(C_S)$ with $s \neq s'$ such that $(s, u), (s', u) \in E(C_S)$. By the distribution of the IDs, we know $ID(s), ID(s') < ID(u)$. Thus by induction hypothesis, we know that for every valid input $e$ there are traces $\pi_2, \pi_3$ representing $(s, u)$ and $(s', u)$ in $X$ that represent the correct value of the edges for input $e$. If $d$ is positive then $\pi$ should be included in $X$ if and only if $\pi_2$ and $\pi_3$ also represent a positive value. Only $\varphi_{and}$ adds traces starting in an AND gate and representing a positive value. $\varphi_{and}$ adds $\pi$ if and only if $\pi, \pi_2$ and $\pi_3$ are positive. If $d$ is negative then $\pi$ should be included in $X$ if and only if $\pi_2$ or $\pi_3$ is negative. $\varphi_{or}$ is the only formula adding traces where $d$ is negative and $u$ is an AND gate. $\varphi_{or}$ adds $\pi$ if it can find a negative trace for the same input and entering $u$. This is the case if and only if $\pi_2$ or $\pi_3$ is negative.

**$u$ is an OR gate:** It holds that $p \vee q \equiv \neg(\neg p \wedge \neg q)$. Every formula adding traces leaving AND gates also adds traces leaving OR gates but with swapped *pos* and *neg* on $\pi, \pi_2, \pi_3$. Therefore, this case is analogous to the case where $u$ is an AND gate.

**$u$ is a NOT gate:** If $u$ is a NOT gate, then there exists exactly one gate $s \in V(C_S)$ such that $(s, u) \in E(C_S)$. We know that $ID(s) < ID(u)$ by the distribution of the IDs. We know by induction hypothesis that for every valid input $e$ there is a trace $\pi_2 \in X$ that represents $(s, u)$. If $d$ is positive, then $\pi$ should be added if and only if $\pi_2$ represents a negative value and if $d$ is negative, then $\pi$ should be added if and only if $\pi_2$ is negative. $\varphi_{not}$ does exactly this.

□

### Phase B

In phase B we collect one trace per input-output pair of $C_S$ in set $Y$. In $X$ the output of $C_S$ for every input is distributed over several traces (one trace per output gate) and we would like to collect them on one trace in a binary representation.

We build two substructures for this process. Similar to phase A both structures consist of $l$ layers. One structure which has four states per layer, named $b_{i,j}$ with $i \in \{1, ..., l\}, j \in \{1, 2, 3, 4\}$. The other structure has two states per layer, which are named $b'_{i,j}$ with $i \in \{1, ..., l\}, j \in \{1, 2\}$.

As in phase A, states $b_{1,j}$ and $b'_{1,j}$ are labeled with $b$ and the initial state of $K$ is connected to them. States $b_{i,1}, b_{i,2}$ and $b'_{i,1}$ with $i \leqslant |e|$ are labeled with atomic proposition *inp* and states $b_{i,1}$ and $b_{i,3}$ are labeled with *outp*. Additionally, all states $b'_{i,j}$ are labeled with atomic proposition *incomplete* and states $b_{l,j}$ are labeled with *end*. An example of a substructure for phase B is given in Fig. 4.4.

→ Fig. 4.4, p. 41

As in phase A, every state is connected to all states in the next layer of their substructure and the states of the last layer of both structures are equipped with self-loops. In contrast to phase A, all states $b_{i,j}$ with $i \in \{1, ..., l-1\}$ have a transition to states $b'_{i+1,1}$ and $b'_{i+1,2}$.

To summarize all traces in the substructure we built for phase B: All traces have a bitstring of length $l$ with proposition *inp* encoded on them. Additionally, they have bitstrings of different lengths encoded with *outp* until they reach a state labeled with *incomplete*. Traces whose *outp* bitstring is of length $l$ never visit a state labeled with *incomplete* but visit a state labeled with *end*.

$\varphi_B$ first computes the intermediate set $Y'$ which contains auxiliary traces as well as the traces we are interested in. $Y$ then consists only of traces from $Y'$ which do not visit the incomplete state. Therefore, $\varphi_B$ looks like this:

$$\varphi_B = (Y', \Upsilon, \varphi'_B \wedge \varphi''_B).(Y, \Upsilon, \forall \pi \in Y'. \square \neg \mathit{incomplete} \rightarrow \pi \triangleright Y).\varphi_C$$

$Y'$ contains for every input $e$ and every prefix $e'$ of the correspondig output one trace $\pi$. $\pi$ has $e$ encoded with *inp* and $e'$ encoded with *outp*.

$$\varphi'_B = \forall \pi \in \mathfrak{G}. \Diamond b \wedge \bigcirc \mathit{incomplete} \rightarrow \pi \triangleright Y'$$

$\varphi'_B$ adds all traces to $Y'$ which immediately visit a state labeled with *incomplete*.

$$\begin{aligned}
\varphi''_B = &\forall \pi_1 \in \mathfrak{G}. \forall \pi_2 \in Y'. \forall \pi_3 \in \mathfrak{G}. \forall \pi_4 \in X. \Diamond b_{\pi_1} \\
&\wedge (\pi_3 =_{AP} \pi_4 \vee \Diamond \mathit{dummy}_{\pi_3}) \\
&\wedge \Box((\mathit{inp}_{\pi_1} \leftrightarrow \mathit{inp}_{\pi_2}) \wedge (\mathit{inp}_{\pi_2} \leftrightarrow \mathit{inp}_{\pi_3})) \\
&\wedge (\mathit{outp}_{\pi_1} \leftrightarrow \mathit{outp}_{\pi_2}) \ \mathcal{U} \ \mathit{incomplete}_{\pi_2} \\
&\wedge \Diamond(\neg \mathit{incomplete}_{\pi_2} \wedge \bigcirc \mathit{incomplete}_{\pi_2} \\
&\quad \wedge \bigcirc(\neg \mathit{incomplete}_{\pi_1} \wedge (\mathit{end}_{\pi_1} \vee \bigcirc \mathit{incomplete}_{\pi_1}) \\
&\quad\quad \wedge \mathit{bit}_{\pi_3} \wedge (\mathit{outp}_{\pi_1} \leftrightarrow \Diamond \mathit{pos}_{\pi_3}))) \\
&\rightarrow \pi_1 \triangleright Y'
\end{aligned}$$

$\varphi''_B$ adds traces to $Y'$ which have one more output bit encoded on them than the traces added in the previous iteration. We quantify over traces $\pi_1, \pi_2, \pi_3$ and $\pi_4$: $\pi_3$ is either a trace representing a dummy output or $\pi_3$ is equal to $\pi_4$ and therefore $\pi_3 \in X$. $\pi_2$ is in $Y'$ from some previous iteration and $\pi_1$ is a trace from the structure relevant for phase B. The only purpose of $\pi_4$ is to allow $\pi_3$ to be either from set $X$ or a dummy trace. We first check whether all three traces have the same input encoded on them and whether $\pi_1$ and $\pi_2$ have the same output encoded until $\pi_2$ enters the *incomplete* states. Then we check the situation in the step where $\pi_2$ first enters the *incomplete* labeled states. $\pi_1$ should continue by one step and then either be incomplete or in a self-loop. If $\pi_1$ encodes $m$ output bits, then $\pi_3$ should represent the edge in $C_S$ going to the $m$-th output gate. $\pi_1$ should set the output bit if and only if $\pi_3$ represents a positive output.

→ Ex. 4.3.1, p. 28

**Example 4.3.2.** We follow Example 4.3.1. To demonstrate how phase B works we focus on traces that specify the behavior of $C_S$ for the input 100. The output of $C_S$ for input 100 is 010. Therefore $Y'$ should contain the following trace:

$$\{\}\{\mathit{inp}\}\{\mathit{outp}\}\{\mathit{end}\}\ldots$$

$\varphi'_B$ is only satisfied if every trace that immediately visits an incomplete state is contained in $Y'$. This also leads to the fact that $Y'$ contains a trace $t'_1$:

$$t'_1 = \{\}\{\mathit{inp}, \mathit{incomplete}\}\{\mathit{incomplete}\}\{\mathit{incomplete}\}\ldots$$

Once $t'_1$ is an element of $Y'$ we can instantiate $\varphi''_B$ as follows:

$$\begin{array}{rcl} \pi_1 = t'_2 & = \{\} & \{inp\} \quad \{incomplete\} \quad \{incomplete\}\ldots \\ \pi_2 = t'_1 & = \{\} & \{inp, incomplete\} \quad \{incomplete\} \quad \{incomplete\}\ldots \\ \pi_3, \pi_4 = t_{16} & = \{\} & \{neg, bit, toid, inp, output\} \quad \{neg, fromid\} \quad \{neg, fromid, toid\}\ldots \end{array}$$

This is the only instantiation, where $\pi_3 =_{AP} \pi_4$, all four traces encode input 100, $\pi_1$ and $\pi_2$ agree on the output until $\pi_2$ visits an incomplete state (which it does immediately) and $\pi_1$ continues $\pi_2$ according to $\pi_3$. That means that at the point where $\pi_2$ visits a state not labeled with *incomplete* and visits states labeled with *incomplete* in the next steps (which is for this instantiation of $\pi_2$ directly at the beginning of the trace) hold the following conditions:

- $\pi_1$ does not visit an incomplete state in the next step. This is satisfied because the second state of $t'_2$ is only labeled with *inp*.
- $\pi_1$ visits an incomplete state in two steps or a state labeled with *end* in the next step. This is satisfied because the third state of $t'_2$ is labeled with *incomplete*.
- $\pi_3$ is labeled with *bit* in the next step. This satisfied because $t_{16}$ represents an edge to the first output gate and is therefore labeled with *bit* in its second state.
- $\pi_1$ is labeled with *outp* in the next state if and only if $\pi_3$ is somewhere labeled with *pos*. This is satisfied because $t'_2$ is not labeled with *outp* anywhere and $t_{16}$ is a negative trace.

The next fixpoint iteration works very similarly.

$$\begin{array}{rcl} \pi_1 = t'_3 & = \{\} & \{inp\} \quad \{outp\} \quad \{incomplete\}\ldots \\ \pi_2 = t'_2 & = \{\} & \{inp\} \quad \{incomplete\} \quad \{incomplete\}\ldots \\ \pi_3, \pi_4 = t_{19} & = \{\} & \{pos, inp, fromid, output\} \quad \{pos, toid, fromid, bit\} \quad \{pos\}\ldots \end{array}$$

For the last iteration, two things change. First, there is no trace in $X$ for which *bit* holds in the third state after the initial state. Therefore $\pi_3$ is instantiated with a trace for a dummy output gate. Second, because $l = 3$ the trace $t'_4$ which is added now to $Y'$

is not incomplete but ends in a state labeled with *end*. The instantiation now looks as follows:

$$\begin{array}{llllll} \pi_1 = t'_4 & = \{\} & \{inp\} & \{outp\} & \{end\}\ldots \\ \pi_2 = t'_3 & = \{\} & \{inp\} & \{outp\} & \{incomplete\}\ldots \\ \pi_3 = & \{\} & \{neg, inp, output, dummy\} & \{neg\} & \{neg, bit\}\ldots \end{array}$$

We can pick an arbitrary trace for $\pi_4$.

Because $t'_4$ is not labeled with *incomplete* anywhere it is part of $Y$. With $t'_4$ there is a trace in $Y$ that for input 100 encodes the correct corresponding output 010. △

**Lemma 8.** *The structure built in phase B can be built in time polynomial in the size of* $C_S$.

*Proof.* The structure built in phase B only depends on $l$. Because $l$ is linear in the size of $C_S$ and the substructure consists of $(2+4)\cdot l$ states it can be built in polynomial time in the size of $C_S$. □

**Lemma 9.** *Let* $\pi$ *be a trace labeled with* $b$*, encoding bitsequence* $e$ *with inp and let* $o$ *be the bitsequence of length* $m$ *encoded on* $\pi$ *with outp before* $\pi$ *visits a state labeled with incomplete. Then holds that* $\pi \in Y'$ *if and only if* $o$ *is a prefix of the output of* $C_S$ *with* $e$ *as an input.*

*Proof.* First, we observe that every trace that is in $Y'$ has to be from the substructures described in phase B because $\varphi'_B$ as well as $\varphi''_B$ check whether a trace is marked with $b$ before adding it to $Y'$. We prove the claim by induction over $m$.

**Case** $m = 0$**:** If $o$ consists of zero bits, it is a prefix for every bit sequence. Therefore every trace encoding an input and zero bits of an output should be in $Y'$. $Y'$ contains all these traces because otherwise, it would not satisfy $\varphi'_B$.

**Case** $0 < m \leqslant l$**:** We show that every trace $\pi$ encoding input $e$ and output $o$ of length $m$ is included in $Y'$ if and only if $o$ is a prefix of the actual output for input $e$ in $C_S$. We know by induction hypothesis that $Y'$ contains a trace $\pi_2$ encoding $m-1$ output bits for input $e$. Additionally, we know from Lem. 7 and the construction of the substructures for phase A that for input $e$ there exists a trace $\pi_3$ where the $m+1$-th step is labeled with *bit*. $\pi_3$ is either in $X$ or labeled with *dummy*. $\pi_3$ is labeled with *dummy* if and only if the $m$-th output gate is a dummy gate. By Lem. 7 $\pi_3$ correctly represents the value of the $m$-th output bit for input $e$. If $\pi_3$ represents a positive value, then the $m$-th bit of the output is a 1. In this case, $\pi$ should be in $Y'$ if and only if it continues $\pi_2$ and visits a state labeled with *outp* where $\pi_2$ first visits a state labeled with *incomplete*. Similarly, if $\pi_3$ is negative, then $\pi$ should be added if it continues $\pi_2$ by a state that is not labeled with *outp*. $\varphi''_B$ specifies exactly this behavior. Trace $\pi$ is added if and only if we find $\pi_2 \in Y'$ and $\pi_3 \in X$

→ Lem. 7, p. 30

such that $\pi$ is labeled with $b$ somewhere, all three traces encode the same input and $\pi$ and $\pi_2$ agree on the output until $\pi_2$ visits an *incomplete* state. The formula $\Diamond(\neg \mathit{incomplete}_{\pi_2} \wedge \bigcirc \mathit{incomplete}_{\pi_2})$ is only satisfied at the $m$-th position because $\pi_2$ encodes $m-1$ bits of output. In the $m+1$-th step several conditions must be true. First, $\pi$ must not be labeled with *incomplete* but has to be labeled *incomplete* in the next step. Alternatively, $\pi$ ends immediately. This is true if and only if $\pi$ encodes exactly one bit more output than $\pi_2$ which means that it encodes $m-1+1=m$ bits of output. Second, $\pi_3$ has to be labeled with *bit* at the $m+1$-th position which is per construction of the substructures in phase A only the case if it represents the output gate responsible for the $m$-th output bit. Finally, $\pi$ has to be labeled with *outp* at position $m+1$ if and only if $\pi_3$ is labeled with *pos* somewhere (after the $m+1$-th position). If $\pi$ visits *outp* in the $m+1$-th position, then the $m$-th bit of the encoded output is true, otherwise it is false. Because all traces in $X$ are labeled with *pos* everywhere if they represent a positive value checking if it is labeled with *pos* after this position is sufficient. Therefore $\pi$ is part of $Y'$ if and only if the $m$-th encoded output bit on $\pi$ has the correct value.

□

**Corollary 10.** *$Y$ contains, for every input $e$ of $C_S$ and its corresponding output $o$, exactly one trace encoding $e$ with inp and $o$ with outp.*

### Phase C

Every input-output pair of $C_S$ encodes an edge in $C$. Therefore phase C is similar to phase A. With the exception that $C$ is encoded in $C_S$ such that every edge $(u,v) \in E(C)$ is represented by two input-output pairs. One pair that identifies $u$ as a predecessor of $v$ and one pair that identifies $v$ as a successor of $u$.

To explicitly annotate the traces from $Y$ with some more atomic propositions we build two substructures in $K$, the resulting structure. Both structures consist of $l$ layers with four states each. We call the $j$-th state in the $i$-th layer of one structure $c_{i,j}$ and the $j$-th state of the $i$-th layer of the other structure $c'_{i,j}$. There are transitions from the initial state of $K$ to states $c_{1,j}$ and $c'_{1,j}$. Additionally, every layer is fully connected to the states of the next layer in their structure and the states in the last layer are equipped with self-loops.

States $c_{i,1}, c_{i,2}, c'_{i,1}$ and $c'_{i,2}$ are labeled with *inp* and states $c_{i,1}, c_{i,3}, c'_{i,1}$ and $c'_{i,3}$ are labeled with *outp*. Additionally, every state $c_{i,j}$ is labeled with *pos* and every state $c'_{i,j}$ is labeled with *neg*. As in the other phases, we mark $c_{1,j}$ and $c'_{1,j}$ with $c$. Because we are interested in the semantics of the input-output pairs, we mark the bit where the encoding of the neighbor starts with atomic proposition *nstart* and the start of the encoded sort with *kstart*. A finished structure can be seen in Fig. 4.5.

→ Fig. 4.5, p. 42

With this structure we have traces binary encoding all possible pairs of bitstrings of length $l$ with *inp* and *outp*. Additionally, every trace is present in these substructures completely labeled with *pos* and completely labeled with *neg*. The traces are marked with atomic propositions *nstart* and *kstart* where the encoding of the neighbor and the sort of gate starts.

Similar to phase A, we maintain a set $Z$ that collects the positive trace for an edge in $C$ if the value on this edge is positive and the negative trace otherwise. Traces that encode input-output pairs of $C_S$ which identify some gate $v$ as a predecessor of some gate $u$, will not be collected in set $Z$ and will be ignored by this phase.

The formula is then similar to the one from phase A but for every gate, its sort is binary encoded after *kstart* and both directions of an edge are represented as a trace. For a given trace we can find out which direction of an edge it represents by looking at the sort of the gate and the encoded neighbor number. As in phase A the formula $\varphi_C$ distincts over the sort of the gates:

$$\varphi_C = (Z, \Upsilon, \varphi_{cconst} \wedge \varphi_{cand} \wedge \varphi_{cor} \wedge \varphi_{cnot}).\varphi_D$$

Similar to phase A, $\varphi_{cand}$ also adds negative traces for OR gates and $\varphi_{cor}$ also adds negative traces for AND gates.

$$\begin{aligned}\varphi_{cconst} =& \forall\pi \in \mathfrak{G}.\forall\pi' \in Y.\Diamond c_\pi \\ & \wedge \Box(inp_\pi \leftrightarrow inp_{\pi'}) \wedge \Box(outp_\pi \leftrightarrow outp_{\pi'}) \\ & \wedge (\Diamond(kstart \wedge \neg outp_\pi \wedge \bigcirc \neg outp_\pi \wedge \bigcirc\bigcirc outp_\pi \wedge \Diamond pos_\pi) \\ & \quad \vee \Diamond(kstart \wedge \neg outp_\pi \wedge \bigcirc outp_\pi \wedge \bigcirc\bigcirc \neg outp_\pi \wedge \Diamond neg_\pi)) \\ & \rightarrow \pi \triangleright Z\end{aligned}$$

Adding the correct traces for TRUE and FALSE gates in $C$ is relatively simple. Because they do not have any meaningful input edges we can simply add all traces representing positive edges involving a TRUE gate and all traces representing a negative edge and a FALSE gate.

The formula quantifies over $\pi$ from the substructure for phase C and $\pi'$ from $Y$. It first checks whether $\pi$ and $\pi'$ have the same input-output pair encoded to make sure that there is actually an edge in $C$ represented by $\pi$. Next, $\pi$ is added to $Z$ if the encoded sort is TRUE and $\pi$ represents a positive edge or if the gate is a FALSE gate and $\pi$ represents a negative edge.

$$
\begin{aligned}
\varphi_{cand} = & \forall \pi_1 \in \mathfrak{G}. \forall \pi_1' \in Y. \forall \pi_2, \pi_3 \in Z. \Diamond c_{\pi_1} \\
& \wedge \Box(inp_{\pi_1} \leftrightarrow inp_{\pi_1'}) \wedge \Box(outp_{\pi_1} \leftrightarrow outp_{\pi_1'}) \\
& \wedge (outp_{\pi_2} \leftrightarrow outp_{\pi_3} \leftrightarrow inp_{\pi_1}) \ \mathcal{U} \ nstart_{\pi_1} \\
& \wedge \neg \Diamond (nstart_{\pi_1} \wedge inp_{\pi_1} \wedge \bigcirc \Box \neg inp_{\pi_1}) \\
& \wedge \neg \Diamond (nstart_{\pi_1} \wedge \neg inp_{\pi_1} \wedge \bigcirc inp_{\pi_1} \wedge \bigcirc \bigcirc \Box \neg inp_{\pi_1}) \\
& \wedge (\Diamond (kstart \wedge \neg outp_{\pi_1} \wedge \bigcirc outp_{\pi_1} \wedge \bigcirc \bigcirc outp_{\pi_1}) \\
& \quad \wedge \Diamond pos_{\pi_1} \wedge \Diamond pos_{\pi_2} \wedge \Diamond pos_{\pi_3} \\
& \quad \vee \Diamond (kstart \wedge outp_{\pi_1} \wedge \bigcirc \neg outp_{\pi_1} \wedge \bigcirc \bigcirc \neg outp_{\pi_1}) \\
& \quad \wedge \Diamond neg_{\pi_1} \wedge \Diamond neg_{\pi_2} \wedge \Diamond neg_{\pi_3}) \\
& \rightarrow \pi_1 \triangleright Z
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{cor} = & \forall \pi_1 \in \mathfrak{G}. \forall \pi_1' \in Y. \forall \pi_2 \in Z. \Diamond c_{\pi_1} \\
& \wedge \Box(inp_{\pi_1} \leftrightarrow inp_{\pi_1'}) \wedge \Box(outp_{\pi_1} \leftrightarrow outp_{\pi_1'}) \\
& \wedge (outp_{\pi_2} \leftrightarrow inp_{\pi_1}) \ \mathcal{U} \ nstart_{\pi_1} \\
& \wedge \neg \Diamond (nstart_{\pi_1} \wedge inp_{\pi_1} \wedge \bigcirc \Box \neg inp_{\pi_1}) \\
& \wedge \neg \Diamond (nstart_{\pi_1} \wedge \neg inp_{\pi_1} \wedge \bigcirc inp_{\pi_1} \wedge \bigcirc \bigcirc \Box \neg inp_{\pi_1}) \\
& \wedge (\Diamond (kstart \wedge \neg outp_{\pi_1} \wedge \bigcirc outp_{\pi_1} \wedge \bigcirc \bigcirc outp_{\pi_1}) \wedge \Diamond neg_{\pi_1} \wedge \Diamond neg_{\pi_2} \\
& \quad \vee \Diamond (kstart \wedge outp_{\pi_1} \wedge \bigcirc \neg outp_{\pi_1} \wedge \bigcirc \bigcirc \neg outp_{\pi_1}) \wedge \Diamond pos_{\pi_1} \wedge \Diamond pos_{\pi_2}) \\
& \rightarrow \pi_1 \triangleright Z
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{cnot} = &\forall \pi_1 \in \mathfrak{G}.\forall \pi_1' \in Y.\forall \pi_2 \in Z.\, \Diamond c_{\pi_1} \\
&\wedge \Box(inp_{\pi_1} \leftrightarrow inp_{\pi_1'}) \wedge \Box(outp_{\pi_1} \leftrightarrow outp_{\pi_1'}) \\
&\wedge (outp_{\pi_2} \leftrightarrow inp_{\pi_1}) \;\mathcal{U}\; nstart_{\pi_1} \\
&\wedge \neg \Diamond(nstart_{\pi_1} \wedge inp_{\pi_1} \wedge \bigcirc\Box inp_{\pi_1}) \\
&\wedge \Diamond(kstart \wedge outp_{\pi_1} \wedge \bigcirc \neg outp_{\pi_1} \wedge \bigcirc\bigcirc outp_{\pi_1}) \\
&\wedge (\Diamond neg_{\pi_1} \wedge \Diamond pos_{\pi_2} \\
&\quad \vee \Diamond pos_{\pi_1} \wedge \Diamond neg_{\pi_2}) \\
&\rightarrow \pi_1 \triangleright Z
\end{aligned}
$$

**Lemma 11.** *The substructure described for phase C can be built in polynomial time in the size of* $C_S$.

*Proof.* The size of the substructure depends only linearly on $l$. Which states should be labeled with *kstart* and *nstart* can be trivially calculated from the input. □

**Lemma 12.** *For every input* $e$ *and corresponding output* $o$ *of* $C_S$ *holds that* $Z$ *contains a trace* $\pi$ *encoding* $e$ *and* $o$. $\pi$ *is labeled with pos if and only if the represented edge of* $C$ *carries a positive value,* $\pi$ *is labeled with neg otherwise.*

*Proof.* We observe that every formula adding a trace $\pi_1$ to $Z$ only adds $\pi_1$ if it is labeled with $c$ at some point and there is a trace $\pi_1' \in Y$ such that $\pi_1$ and $\pi_1'$ encode the same input and output. Therefore every trace in $Z$ is from the substructure for phase C and every trace in $Z$ encodes a valid input-output pair of $C_S$. Further, a trace $\pi$ represents an edge in $C$ which goes from a gate $u \in V(C)$ to a gate $v \in V(C)$. The ID of $u$ is encoded on $\pi$ with *inp* until $\pi$ visits a state labeled with *nstart* and similarly the ID of $v$ is encoded on $\pi$ with *outp* until $\pi$ visits a state labeled *kstart*. Therefore holds the following: Let $\pi_1$ encode $(u, v) \in E(C)$ and $\pi_2$ encode $(s, t) \in E(C)$. $(outp_{\pi_2} \leftrightarrow inp_{\pi_1}) \;\mathcal{U}\; nstart_{\pi_1}$ is satisfied if and only if $t = u$.

Additionally, $Z$ should only contain traces representing the edge $(u, v)$ where $v$ is the $k$-th neighbor (and successor) of $u$ and not the traces representing the fact that $u$ is the first or second neighbor (and predecessor) of $v$. By the definition of the Succinct Circuit Value Problem, we know that the first two neighbors of AND and OR gates are its predecessors and the first neighbor of a NOT gate is its predecessor. Therefore $\varphi_{cand}$ and $\varphi_{cor}$ require that a trace $\pi_1$ can only be added to $Z$ if it does not encode a one or a two after *nstart*. Similarly $\varphi_{cnot}$ requires that $\pi_1$ can only be added if it does not encode a one after *nstart*. This suffices for $Z$ to only contain traces with information about successors.

The only difference to phase A is now that the traces added into $Z$ represent edges from $C$ and not $C_S$ and that every trace encoding edge $(u, v)$ contains $sort(u)$ and not

$sort(v)$. The formulas check for the $sort(u)$ by looking at the state where *kstart* holds and the next two states. Therefore dummy outputs that were added to $C_S$ in phase A do not have any effect here. For the rest of the proof we do well-founded induction over the gates of $C$ with the relation which contains a pair of gates $(s,t) \in V(C)^2$ if and only if $(s,t) \in E(C)$. We now case distinct over the $sort(u)$. Every case is analogous to the cases in phase A except the case for which $u$ is a TRUE or FALSE gate.

**$u$ is a constant gate:** If $u$ is a gate for constant true, then all edges leaving $u$ carry a positive value. Therefore a trace representing an edge leaving $u$ should be included if and only if it is labeled with *pos*. Similarly, if $u$ is a FALSE gate a trace representing an edge leaving $u$ should be contained if and only if it is labeled with *neg*. $Z$ conatins exactly the described traces because it would not satisfy $\varphi_{const}$ otherwise.

□

After building set $Z$ we can determine the output of $C$ by looking for a gate with no output. By definition of the Succinct Circuit Value Problem, a gate with no output has its first output connected to a gate with ID 0. Therefore $\varphi_D$ checks whether there exists a trace $\pi \in Z$ which represents an edge that carries a positive value and is connected to a gate with ID zero. Additionally, $\pi$ needs to be the first output of a gate for which $\varphi_D$ case distincts over the sort of the gate from which $\pi$ is coming and encoded neighbor number.

$$
\begin{aligned}
\varphi_D = & \exists \pi \in Z. \Diamond pos \wedge \neg outp \ \mathcal{U} \ kstart \\
& \wedge (\Diamond(nstart_\pi \wedge inp_\pi \wedge \bigcirc inp_\pi \wedge \bigcirc\bigcirc\Box\neg inp) \\
& \quad \wedge \Diamond(kstart \wedge outp_\pi \wedge \bigcirc\neg outp_\pi \wedge \bigcirc\bigcirc\neg outp_\pi) \\
& \vee \Diamond(nstart_\pi \wedge inp_\pi \wedge \bigcirc inp_\pi \wedge \bigcirc\bigcirc\Box\neg inp) \\
& \quad \wedge \Diamond(kstart \wedge \neg outp_\pi \wedge \bigcirc outp_\pi \wedge \bigcirc\bigcirc outp_\pi) \\
& \vee \Diamond(nstart_\pi \wedge inp_\pi \wedge \bigcirc\Box\neg inp) \\
& \quad \wedge \Diamond(kstart \wedge outp_\pi \wedge \bigcirc\neg outp_\pi \wedge \bigcirc\bigcirc outp_\pi) \\
& \vee \Diamond(nstart_\pi \wedge \Box\neg inp) \\
& \quad \wedge \Diamond(kstart \wedge \neg outp_\pi \wedge \bigcirc\neg outp_\pi \wedge \bigcirc\bigcirc outp_\pi) \\
& \vee \Diamond(nstart_\pi \wedge \Box\neg inp) \\
& \quad \wedge \Diamond(kstart \wedge \neg outp_\pi \wedge \bigcirc outp_\pi \wedge \bigcirc\bigcirc\neg outp_\pi))
\end{aligned}
$$

**Lemma 13.** $\mathsf{K}$ *is acyclic*

*Proof.* For every phase we built substructures consisting of layers and every layer is only connected to the next layer. The only layers that are not connected to the next layer are the last layers which are equipped with self-loops that satisfy our definition of acyclic structures. □

**Theorem 14.** *MC[Fixpoint, Acyclic] is EXP-complete*

*Proof.* The answer to the Succinct Circuit Value Problem is true if and only if there exists a gate $u \in V(C)$ with no other gate $v \in V(C)$ such that $(u, v) \in E(C)$ and the output of
→ Lem. 12, p. 38
gate $u$ is true. We know by Lem. 12 that $Z$ contains a trace labeled with *pos* and encoding the $\mathrm{ID}(u)$ with *inp* for some gate $u$ if and only if the output of gate $u$ is true. By the definition of the Succinct Circuit Value Problem follows that if the $i+1$-th neighbor of a gate with $i$ inputs has ID zero then the gate has no successor and is the output of $C$. Note that the number of inputs a gate has is determined by its sort. Therefore the answer to the Succinct Circuit Value Problem is true if and only if $Z$ contains a trace $\pi$ that is labeled with *pos* and represents an edge going to a gate with ID zero which is the $i+1$-th neighbor of a gate with $i$ inputs. By the EXP-completeness of the Succinct Circuit Value Problem [Pap94] follows the EXP-hardness of MC[Fixpoint, Acyclic].
→ Lem. 5, p. 22
From the EXP-hardness of MC[Fixpoint, Acyclic] and Lem. 5 follows that MC[Fixpoint, Acyclic] is EXP-complete. □

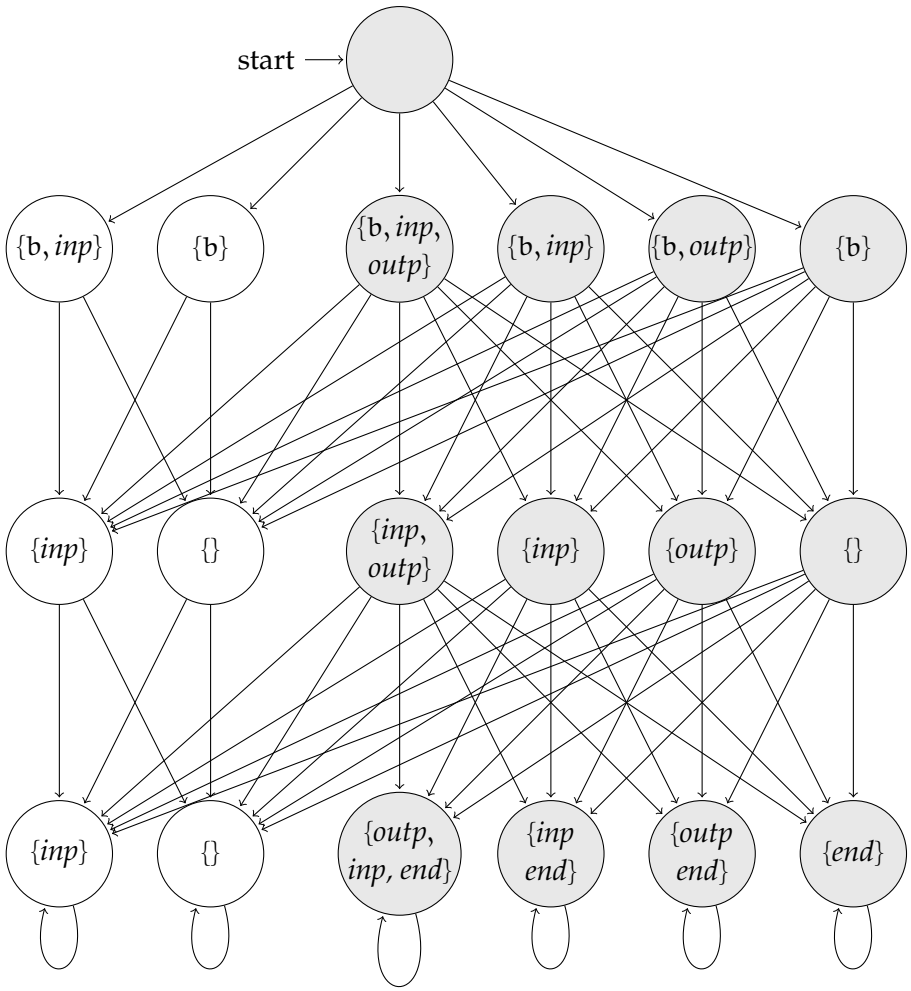



Figure 4.4: Substructures for phase B with $\mathfrak{l} = |e| = 3$. White states are additionally labeled with *incomplete*.

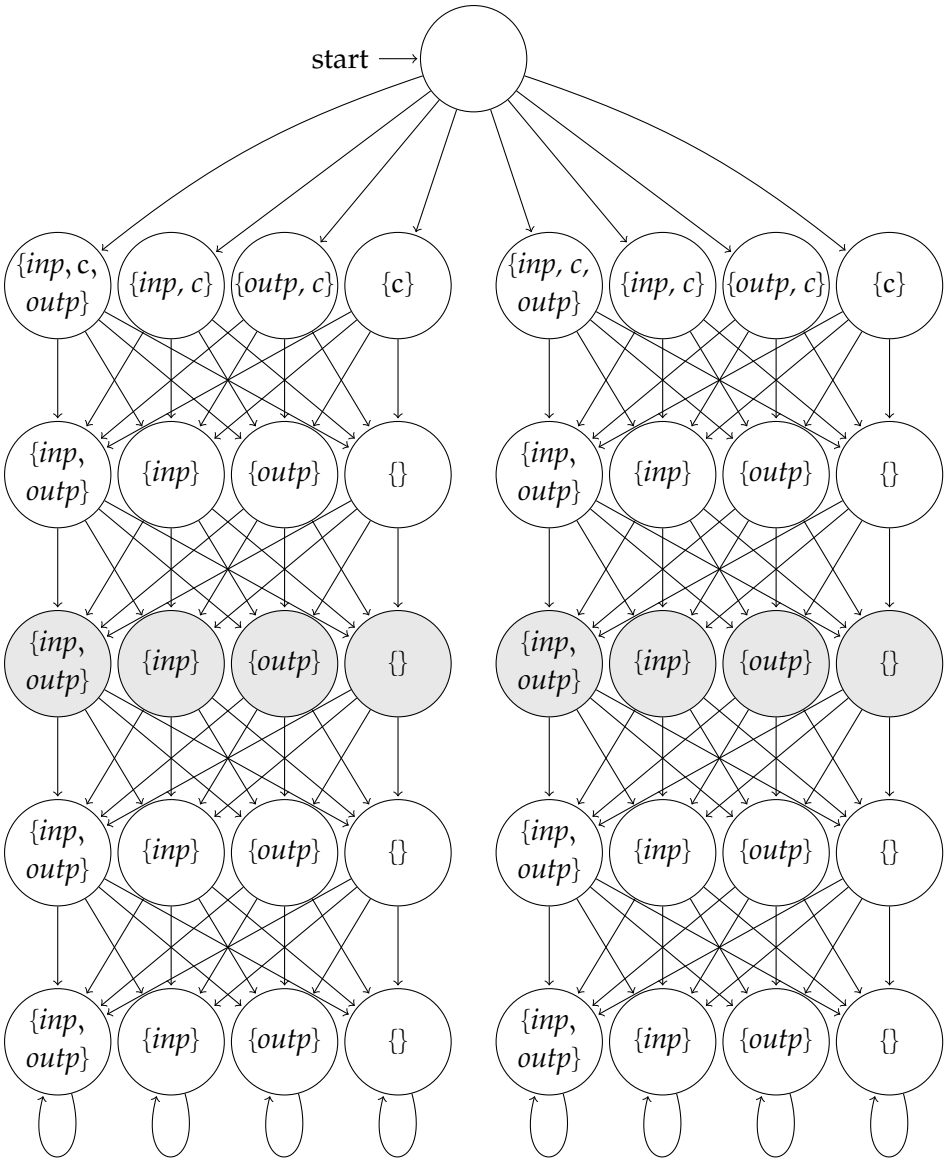



Figure 4.5: The substructure built for phase C with $l = 5$ and two bits to encode a gate in C. The states on the left are additionally labeled with *pos* and the states on the right are additionally labeled with *neg*. The atomic propositions *kstart* and *nstart* only occur on the grey states.

# 5 The Complexity of MC[HYPER$^2$LTL]

In this chapter, we analyze the complexity of the model checking problem for Hyper$^2$LTL. First, we show that MC[HYPER$^2$LTL, TREE] is complete in the respective classes of the polynomial hierarchy and in Sect. 5.2 we show that MC[HYPER$^2$LTL, ACYCLIC] is contained in the exponential hierarchy. We show in Sect. 5.3 that MC[FIXPOINT, TREE] as well as MC[HYPER$^2$LTL, TREE] is PSPACE-complete in the combined input size consisting of Kripke structure and formula.

## 5.1 The Complexity of MC[HYPER$^2$LTL, TREE]

We show the $\Sigma^p_{k+1}$ resp. $\Pi^p_{k+1}$-completeness by first showing containment in the respective classes. Then we prove hardness in these classes by reducing from the QBF problem.

**Lemma 15.** *MC[HYPER$^2$LTL, TREE] with $k$ second-order quantifier alternations is in $\Sigma^p_{k+1}$ if the outermost second-order quantifier is existential and in $\Pi^p_{k+1}$ if the outermost second-order quantifier is universal.*

*Proof.* Given Hyper$^2$LTL formula $\varphi$ and Kripke structure $K$. Let $n$ be the number of states of $K$ and let $q$ be the number of quantifiers in $\varphi$. Because $\varphi$ is not part of the input, $q$ is assumed to be constant. Instantiations of quantifiers (traces resp. sets of traces) are saved by marking the corresponding leaf resp. leaves. We collect a set $I$ of all assignments that need to be checked. This set is calculated from the outermost quantifier to the innermost quantifier. Every quantifier, binding a variable $x$, expands $I$ such that all assignments in $I$ also assign $x$. For first-order quantifiers, we add up to $n$ possible assignments for each entry already in $I$ and for second-order quantifiers we guess a set for each instantiation already in $I$. This means that we have in the end at most $n^q$ assignments in $I$ that we need to verify against the quantifier free formula. We will prove the claim by structural induction over $\varphi$. The induction hypothesis here is as follows: Let second-order quantifier assignments be already contained in $I$ such that a formula $\varphi'$ with $k-1$ second-order quantifier alternations remains. Verifying that under the assignments in $I$ $\varphi'$ is satisfied, is in $\Sigma^p_k$ if the outmost second-order quantification in $\varphi'$ is existential and in $\Pi^p_k$ otherwise.

**First-order quantification** Let $x$ be the variable bound by the first-order quantifier. For each assignment $a \in I$ and every possible instantiation $b$ of $x$, we replace $a$ with a

new assignment that is equal to $a$ and additionally assigns $x$ to $b$. Because there are at most $n$ possible instantiations of $x$, the size of $I$ can increase by a factor of $n$. The inner formula is then evaluated with the updated set $I$. Because $I$ can be updated in time polynomial to the size of the original size of $I$, which is polynomial in $K$, every first-order quantifier only adds polynomial time to the complexity.

**Existential second-order quantification** For each assignment $a \in I$ we nondeterministically guess instantiation for all existential second-order quantifiers at the beginning of $\varphi$. The guessed instantiations are then added to assignment $a$. If there is a first-order quantifier between the existential second-order quantifiers, we add new instantiations to $I$ as described above. A set of traces consists of at most $n$ traces and the number of instantiations we need to guess is less or equal to $q \cdot |I| \leqslant q \cdot n^q$. Therefore, we have to guess at most polynomially many bits that correspond to sets of traces, each added to one assignment in $I$. By the induction hypothesis, the correctness of the assignments in $I$ can be verified in $\Pi^p_k$. Existentially guessing polynomially bits in $n$ and verifying them in $\Pi^p_k$ has a complexity of $\Sigma^p_{k+1}$.

**Universal second-order quantification** For each assignment $a \in I$ we universally nondeterministically guess instantiation for all universal second-order quantifiers at the beginning of $\varphi$. The guessed instantiations are then added to assignment $a$. If there is a first-order quantifier between the universal second-order quantifiers, we add new instantiations to $I$ as described above. A set of traces consists of at most $n$ traces and the number of instantiations we need to guess is less or equal to $q \cdot |I| \leqslant q \cdot n^q$. Therefore, we have to guess at most polynomially many bits that correspond to sets of traces, each added to one assignment in $I$. By the induction hypothesis, the correctness of the instantiations can be verified in $\Sigma^p_k$. Universally guessing polynomially many bits in $n$ and verifying it in $\Sigma^p_k$ has a complexity of $\Pi^p_{k+1}$.

**Quantifier free** When $I$ contains assignments that assign a value to every variable, we can evaluate the inner formula. We iterate over every assignment in $I$ and evaluate the quantifier free formula under this assignment. From the results of these evaluations and the information whether a first-order variable was universally or existentially quantified, we can calculate whether or not $K$ satisfies $\varphi$. This needs polynomial time because there are at most $n^q$ assignments in $I$ and we need polynomial time in $n$ to evaluate the quantifier free formula for each assignment. Additionally, deciding whether or not $K$ satisfies $\varphi$ can be done by iterating over each element of $I$ and its corresponding result.

We do not need more than polynomial time, because $q$ is constant in the size of the input. Additionally, we do not change between universal and existential guessing more than $k$ times. The complexity of the algorithm is therefore in the polynomial hierarchy. □

### 5.1.1 Lower bound

After showing the containment in the classes $\Sigma^p_{k+1}$ and $\Pi^p_{k+1}$ it remains to show that the problem is $\Sigma^p_{k+1}$-hard resp. $\Pi^p_{k+1}$-hard. To do this, we reduce from the Quantified Boolean Formula Problem (QBF).

**Definition 5.1** (Quantified Boolean Formula)

Given $k, m_1, \ldots, m_{k+1} \in \mathbb{N}$ and formula $\psi$ which is in one of the two following forms. We say that $\psi$ starts with an existential quantifier iff $\psi$ is of the following form:

$$\exists x_{1,1} \ldots \exists x_{1,m_1} . \forall x_{2,1} \ldots \forall x_{2,m_2} \ldots \mathbb{Q} x_{k+1,1} \ldots \mathbb{Q} x_{k+1,m_{k+1}} E$$

where $\mathbb{Q} = \exists$ if $k$ is even and $\mathbb{Q} = \forall$ if $k$ is odd and $E$ is an arbitrary Boolean formula over the variables $x_{1,1}, \ldots, x_{k+1,m_{k+1}}$. We say that $\psi$ starts with a universal quantifier iff $\psi$ is of the following form:

$$\forall x_{1,1} \ldots \forall x_{1,m_1} . \exists x_{2,1} \ldots \exists x_{2,m_2} \ldots \mathbb{Q} x_{k+1,1} \ldots \mathbb{Q} x_{k+1,m_{k+1}} E$$

where $\mathbb{Q} = \exists$ if $k$ is odd and $\mathbb{Q} = \forall$ if $k$ is even and $E$ is an arbitrary Boolean formula over the variables $x_{1,1}, \ldots, x_{k+1,m_{k+1}}$.

We say that $\psi$ has $k$ quantifier alternations. The QBF problem is then to decide whether $\psi$ holds.

The QBF problem where the number of quantifier alternations is bounded by $k$ is $\Sigma^p_{k+1}$-complete if the formula starts with an existential quantification and it is $\Pi^p_{k+1}$-complete if the formula starts with a universal quantification [GJ79]. That means that the QBF problem where the number of quantifier alternations ($k$) is restricted behaves similarly to MC[HYPER$^2$LTL, TREE] where the number of second-order quantifier alternations is restricted. We will use this property in the following where we reduce QBF with restricted $k$ to MC[HYPER$^2$LTL, TREE] with $k$ second-order quantifier alternations.

Given such a QBF formula $\psi$ with $k$ quantifier alternations, we build a tree-shaped Kripke structure $K$ in polynomial time. Additionally, we build a Hyper$^2$LTL formula $\varphi$ which only depends on $k$ and the first quantifier of $\psi$.

The Kripke structure $K$ consists of an initial state that is connected to several branches, which do not branch further. Every branch has length $l$, where $l$ is the sum of the number of variables and Boolean operators in $\psi$. Note that $k \leqslant l$.

Every variable $x_{i,j}$ and subformula $e$ in $\psi$ is associated with a number $N(x_{i,j}), N(e)$. We number the variables and subexpressions subsequently, such that the number of every variable is smaller than the number of every operator. When reasoning about a Boolean operator $\otimes$, $N(\otimes)$ denotes the number of the subexpression where $\otimes$ is the uppermost operator.

For every variable $x_{i,j}$, $K$ contains two branches. One branch is marked with *pos* in the $N(x_{i,j})$-th state and the other branch is marked with *neg* in the $N(x_{i,j})$-th state.

For every subexpression $E_1 \otimes E_2$ in $y$, where $\otimes$ is an arbitrary binary operator, $K$ contains three branches.

If $\otimes = \wedge$, then its first branch is labeled with *pos* at the $N(\otimes)$-th state and with *epos* in the $N(E_1)$-th as well as with *epos'* in the $N(E_2)$-th state. The second branch is labeled with *neg* in the $N(\otimes)$-th state and with *eneg* in the $N(E_1)$-th state. The third branch is labeled with *neg* in the $N(\otimes)$-th state and with *eneg* in the $N(E_2)$-th state.

If $\otimes = \vee$, then its first branch is labeled with *neg* at the $N(\otimes)$-th state and with *eneg* in the $N(E_1)$-th as well as with *eneg'* in the $N(E_2)$-th state. The second branch is labeled with *pos* in the $N(\otimes)$-th state and with *epos* in the $N(E_1)$-th state. The third branch is labeled with *pos* in the $N(\otimes)$-th state and with *epos* in the $N(E_2)$-th state.

For every negated subexpression $E = \neg E_1$ in $y$, $K$ contains two branches. The first branch is labeled with *pos* in the $N(E)$-th state and *eneg* in the $N(E_1)$-th state. The second branch is labeled with *neg* in the $N(E)$-th state and *epos* in the $N(E_1)$-th state.

All branches of the outermost Boolean operator of $y$ are labeled with $f$ in the first state.

$K$ contains $k+1$ additional branches. The $i$-th branch is marked at the $i$-th position with atomic proposition $q$. $m_i$ of the states of this branch are marked with $\nu$. The $j$-th state of the $i$-th branch is marked with $\nu$ if and only if there exists $a \leqslant m_i$ such that $x_{i,a}$ is associated with $j$.

$\varphi$ consists of $k+2$ second-order quantifications. We will divide $\varphi$ into subformulas that each contain only one second-order quantification.

$$\varphi = \varphi_1$$

We first give $\varphi_1$ to $\varphi_{k+1}$. $\varphi_i$ with $i < k+2$ is of the form:

$$\begin{aligned}
&\mathbb{Q}X_i.(\forall \pi \in \mathfrak{G}.\exists \pi' \in X.\exists! \pi'' \in X.\bigcirc^i q_{\pi'} \\
&\quad \wedge (\square(\neg \mathit{epos}_\pi \wedge \neg \mathit{eneg}_\pi \wedge \neg q_\pi) \rightarrow \diamondsuit(\nu_{\pi'} \wedge (\mathit{pos}_\pi \vee \mathit{neg}_\pi) \rightarrow (\mathit{pos}_{\pi''} \vee \mathit{neg}_{\pi''})))) \\
&\oplus \varphi_{i+1}
\end{aligned}$$

If variables $x_{i,1}$ to $x_{i,m_i}$ are quantified universally in $y$, then $\mathbb{Q} = \forall, \oplus = \rightarrow$. If they are quantified existentially in $y$, then $\mathbb{Q} = \exists, \oplus = \wedge$.

$\varphi_{k+2}$ should not introduce another quantifier alternation. Therefore, its quantifier depends on the quantifier of $\varphi_{k+1}$.

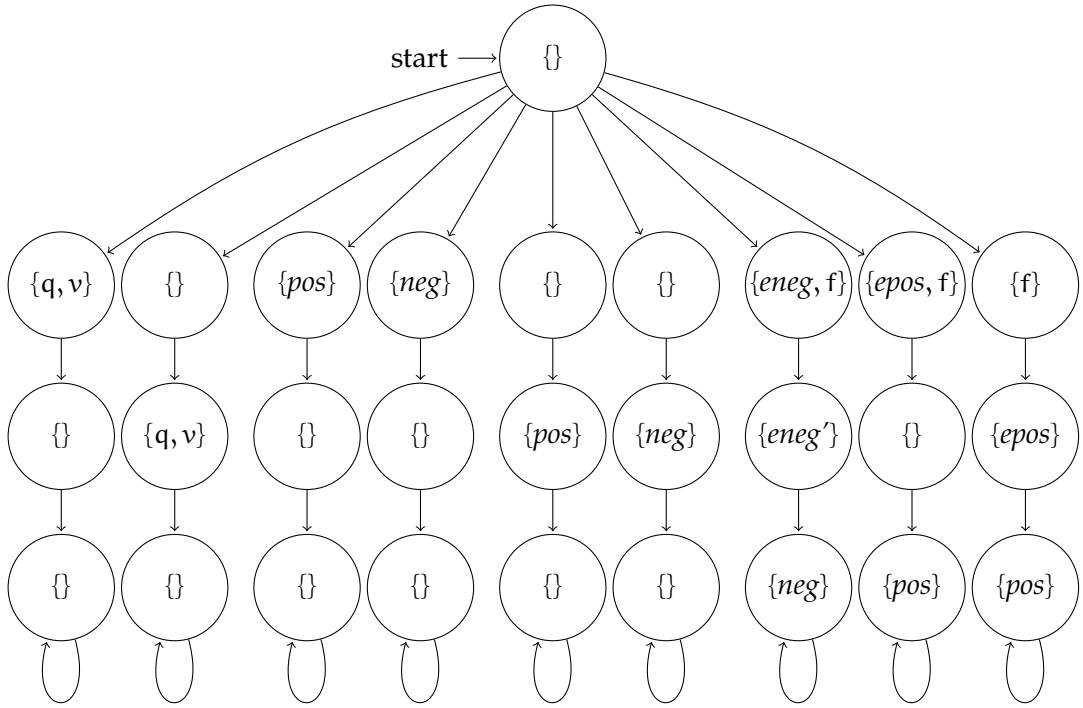



Figure 5.1: Kripke structure built for the QBF formula $\forall x_1.\exists x_2.x_1 \vee x_2$

$$\varphi_{k+2} = \mathbb{Q}Z.(\forall \pi_1 \in X_1 \ldots \forall \pi_{k+1} \in X_{k+1}.\pi_1 \triangleright Z \wedge \cdots \wedge \pi_{k+1} \triangleright Z) \quad (1)$$

$$\wedge (\exists \pi \in Z.\Diamond f_\pi) \quad (2)$$

$$\wedge (\forall \pi \in Z.\exists \pi' \in Z.\exists \pi'' \in Z.\Box((epos_\pi \leftrightarrow pos_{\pi'}) \wedge (eneg_\pi \leftrightarrow neg_{\pi'}) \quad (3)$$

$$\wedge (epos'_\pi \leftrightarrow pos_{\pi''}) \wedge (eneg'_\pi \leftrightarrow neg_{\pi''}))) \quad (4)$$

$$\oplus \exists \pi \in Z.\Diamond f_\pi \wedge \Diamond pos_\pi \quad (5)$$

If $\varphi_{k+1}$ is universally quantified, then $\mathbb{Q} = \forall$ and $\oplus = \rightarrow$. If $\varphi_{k+1}$ is existentially quantified, then $\mathbb{Q} = \exists$ and $\oplus = \wedge$.

Note that $\varphi$ is not given in formal Hyper$^2$LTL syntax but can be easily transformed into it.

**Example 5.1.1.** Consider the case where $y = \forall x_1.\exists x_2.x_1 \vee x_2$. $y$ has one quantifier alternation, such that $k = 1$. Additionally, $y$ is true and therefore the model checking problem should be satisfied. The Kripke structure built by the reduction can be seen in Fig. 5.1.

$\varphi$ for $k = 1$ starting with a universal quantifier, quantifies over the sets $X_1, X_2$ and $Z$. $X_1$ is universally quantified and $X_2$ as well as $Z$ are existentially quantified. By existentially quantifying over $Z$, $\varphi$ only contains one quantifier alternation.

The two possible instantiations for $X_1$ are:

| | |
|---|---|
| $\{\}\{q, v\}\{\}\{\}\dots$ | $\{\}\{q, v\}\{\}\{\}\dots$ |
| $\{\}\{pos\}\{\}\{\}\dots$ | $\{\}\{neg\}\{\}\{\}\dots$ |

For $X_2$ are the possible instantiations:

| | |
|---|---|
| $\{\}\{\}\{q, v\}\{\}\dots$ | $\{\}\{\}\{q, v\}\{\}\dots$ |
| $\{\}\{\}\{pos\}\{\}\dots$ | $\{\}\{\}\{neg\}\{\}\dots$ |

There exists a set $Z$ satisfying the condition in $\varphi_{k+2}$ resp. $\varphi_3$ if the trace $\{\}\{\}\{pos\}\{\}\dots$ is an element of either $X_1$ or $X_2$. Because for both instantiations of $X_1$ there exists an instantiation of $X_2$ which contains $\{\}\{\}\{pos\}\{\}\dots$, the model checking problem is satisfied. $\triangle$

**Lemma 16.** $K \models \varphi$ *if and only if* $y$ *is true.*

*Proof.* We first argue that an instantiation of $X_i$ with $i \leqslant k + 1$ satisfying $\varphi_i$ represents an assignment to all variables $x_{i,j}$ where $j \leqslant m_i$. There exists only one trace $\pi'$ in $K$ satisfying $\bigcirc^i q_{\pi'}$. $\pi'$ is the only possible instantiation for $\pi'$ in $\varphi_i$. It is marked with $v$ at every position which represents a variable $x_{i,j}$. An instantiation now contains, for each variable, exactly one trace that has *pos* or *neg* at the respective position. All traces representing variables satisfy $\square(\neg epos_\pi \wedge \neg eneg_\pi \wedge \neg q_\pi)$. The traces representing variables $x_{i,j}$ with $j \leqslant m_i$ additionally satisfy $\diamondsuit(v_{\pi'} \wedge (pos_\pi \vee neg_v))$. $\varphi_i$ is satisfied if and only if, for each trace $\pi$ representing variable $x_{i,j}$, there is exactly one trace $\pi'' \in X$ that represents the same variable.

By construction of $K$ as well as $\varphi$ we quantify these assignments exactly as $y$ does. It remains to show that under fixed instantiations for $X_1, \dots, X_{k+1}$, $\varphi_{k+2}$ is satisfied if and only if $E$ is satisfied under the assignment represented by $X_1, \dots, X_{k+1}$.

First, we note that some trace $\pi$ can only be part of $Z$ if there are traces $\pi', \pi''$ that satisfy the condition in lines 3 and 4 of $\varphi_{k+2}$. By the construction of $K$ and $\varphi_{k+2}$, this is the case if and only if the subexpressions represented by $\pi'$ and $\pi''$ have the value represented by *epos* and *eneg*.

We start by showing that every possible instantiation of $Z$ contains one trace per quantifier free subexpression in $y$. We call this largest quantifier free subexpression of $y$ $E$. We do this by well-founded induction over the relation that relates every subexpression of $E$ with its one resp. two largest subexpressions. In the case that the

subexpression is a variable, it is not related to another subexpression. The base case here is $E$. By construction of $K$, only traces representing $E$ are marked with $f$. Additionally, $\varphi_{k+2}$ enforces that there is at least one trace in $Z$ that is marked with $f$. Therefore, $Z$ contains a trace for $E$. The induction step follows from the fact that a trace $\pi$ can only be contained in $Z$ if there are two traces $\pi', \pi'' \in Z$ satisfying the condition in lines 3 and 4. This is by construction of $K$ the case only if $\pi'$ and $\pi''$ represent the one resp. two largest subexpressions of the expression represented by $\pi$. In the case where $\pi$ represents a variable, $\pi'$ as well as $\pi''$ can be instantiated with a trace labeled with $q$ because then $\pi'$ and $\pi''$ are not labeled with *pos* or *neg*.

Next, we prove by structural induction over $E$ that every possible instantiation of $Z$ contains a trace $\pi$ labeled with *pos* at position $j$ if and only if the operator resp. variable associated with $j$ evaluates to true. Instantiations for $Z$ contain the *neg* trace otherwise.

$x_{a,b}$ A variable is true iff it is assigned to true. Because $Z$ is a superset of $X_1 \cup \cdots \cup X_{k+1}$, $Z$ contains the trace representing the value the variable is assigned to.

$E_1 \wedge E_2$ The positive trace $\pi$ can be contained in $Z$, if there exist traces $\pi', \pi''$ that are marked with *pos* where $\pi$ is marked with *epos* or *epos'*. By the construction of $K$, we know that $\pi'$ and $\pi''$ represent $E_1$ and $E_2$. By the induction hypothesis follows that $\pi'$ and $\pi''$ are only in $Z$ if and only if $E_1$ as well as $E_2$ are true. In this case, $E_1 \wedge E_2$ is also true and $\pi$ should be part of $Z$.

If $\pi$ is one of the negative traces it can only be included if there exist $\pi', \pi'' \in Z$ such that at least one of them is marked with *neg* where $\pi$ is marked with *eneg*. We know that this trace represents either $E_1$ or $E_2$ by the construction of $K$. By induction hypothesis, we know that these two traces only exist in $Z$ if either $E_1$ or $E_2$ is negative. In this case, $E_1 \wedge E_2$ is false and $\pi$ should be included in $Z$.

If only one of $E_1$ or $E_2$ is negative, we can always instantiate $\pi''$ with a trace that is neither marked with *pos* nor with *neg*.

$E_1 \vee E_2$ Analogous to $E_1 \wedge E_2$.

$\neg E_1$ If $\pi$ is the negative trace, it is marked with *epos* at the position that represents $E_1$. $\pi$ can only be part of $Z$ if there exist traces $\pi', \pi'' \in Z$ such that $\pi'$ is marked with *pos* exactly where $\pi$ is marked with *epos*. Additionally, $\pi''$ has to be marked with *pos* or *neg* exactly where $\pi$ is marked with *epos'* or *eneg'*. $\pi''$ can be instantiated with some trace labeled neither with *pos* nor *neg*. By the induction hypothesis, trace $\pi'$ with the described properties only exists if $E_1$ is true. Therefore, the negative trace $\pi$ can only be included in $Z$ if $\neg E_1$ is false.

If $\pi$ is the positive trace, the situation is analogous.

Only the traces representing $E$ are labeled with $f$. Therefore, a trace $\pi$ labeled with $f$ as well as *pos* is contained in $Z$ if and only if $E$ is true under the assignment represented by $X_1, \ldots, X_{k+1}$. □

**Lemma 17.** *The reduction of Lem. 16 can be done in time that is polynomial in the size of* $y$.

*Proof.* $K$ consists of branches of length $l$, which is polynomial in the size of $y$. Additionally, $K$ contains only a constant amount of branches per operator and variable in $y$. Therefore, $K$ can be built in polynomial time.

$\varphi$ only depends on the number of quantifier alternations in $y$. These are linear in the size of $y$. □

**Theorem 18.** *MC[Hyper$^2$LTL, Tree] is* $\Sigma^p_{k+1}$*-complete if the given formula has* $k$ *quantifier alternations and the outermost second-order quantifier is existential. MC[Hyper$^2$LTL, Tree] is* $\Pi^p_{k+1}$*-complete if the given formula has* $k$ *quantifier alternations and the outermost second-order quantifier is universal.*

*Proof.* The Quantified Boolean Formula Problem with $k$ quantifier alternations is $\Sigma^p_{k+1}$-complete if the first quantifier is existential and $\Pi^p_{k+1}$-complete otherwise [GJ79]. By the reduction (Lem. 16, Lem. 17) holds for every $k$, that MC[Hyper$^2$LTL, Tree] with $k$ quantifier alternations is $\Sigma^p_{k+1}$-hard if the formula starts with an existential quantifier and $\Pi^p_{k+1}$-hard otherwise.

By Lem. 15 the completeness in the respective complexity classes follows. □

**Corollary 19.** *MC[Hyper$^2$LTL, Tree] with an unrestricted number of second-order quantifier alternations is in PSPACE.*

## 5.2 The Complexity of MC[HYPER$^2$LTL, ACYCLIC]

We show that MC[HYPER$^2$LTL, ACYCLIC] where the given formula has at most $k$ second-order quantifier alternations is in $\Sigma^{EXP}_{k+1}$ if the outermost second-order quantifier is existential and in $\Pi^{EXP}_{k+1}$ otherwise. From this follows that Hyper$^2$LTL model checking on acyclic models is decidable as well as that it is in EXPSPACE.

**Lemma 20.** *MC[HYPER$^2$LTL, ACYCLIC] with $k$ second-order quantifier alternations is in $\Sigma^{EXP}_{k+1}$ if the outermost second-order quantifier is existential and in $\Pi^{EXP}_{k+1}$ if the outermost second-order quantifier is universal.*

*Proof.* Let $K$ be the given acyclic model and let $\varphi$ be the given formula with $k$ second-order quantifier alternations. The size of $\varphi$ as well as $k$ are constant.

We evaluate $\varphi$ from the outermost quantifier to the innermost quantifier. To save a trace we mark all states that it contains. This requires space polynomial in the size of $K$. To save a set of traces requires exponential space in the size of $K$ because $K$ can contain up to exponentially many traces.

We proceed exactly as in Lem. 15. Guessing a second-order quantifier here needs time exponential in the size of $K$ because we need to guess exponentially many bits. Additionally, there are exponentially many instantiations for every first-order quantifier, such that exponential time in the size of $K$ is required to evaluate it.

→ Lem. 15, p. 43

The proof is then analogous to the proof of Lem. 15 with the exception that we are in $\Sigma^{EXP}_{k+1}$ and $\Pi^{EXP}_{k+1}$ instead of $\Sigma^{P}_{k+1}$ and $\Sigma^{P}_{k+1}$. □

**Corollary 21.** *MC[HYPER$^2$LTL, ACYCLIC] with an unrestricted number of second-order quantifier alternations is in EXPSPACE.*

## 5.3 Complexity in the Size of the Model and Formula

We prove in this chapter, that MC[Hyper$^2$LTL, Tree] as well as MC[Fixpoint, Tree] is PSPACE-complete in the combined size of the Kripke structure and formula.

**Lemma 22.** *MC[Hyper$^2$LTL, Tree] is in PSPACE.*

*Proof.* Given Hyper$^2$LTL formula $\varphi$ and tree-shaped Kripke structure $K$. Let $n$ be the number of states of $K$, let $m$ be the size of $\varphi$ and let $q$ be the number of quantifiers in $\varphi$. We evaluate $\varphi$ recursively from the outermost quantifier to the innermost quantifier. To evaluate a quantifier we iterate over every possible instantiation and recursively evaluate its inner formula. Instantiations of quantifiers (traces resp. sets of traces) are saved by marking the corresponding leaf resp. leaves.

When all traces are fixed the inner, quantifier free formula can be checked in polynomial space [BF18]. To every point in time, there are no more than $q$ quantifier instantiations saved. Saving a set of traces needs $\mathcal{O}(n \cdot \log(n))$ space, because there are at most $n$ traces in $K$ and saving a trace needs $\mathcal{O}(\log n)$ space. The space required to find the next set of traces or the next trace is polynomial in $n$ and constant in $m$. Therefore, it is not more than $\mathcal{O}(q \cdot n \cdot \log(n))$ space required, which is polynomial in the combined size of structure and formula. □

**Theorem 23.** *MC[Fixpoint, Tree] is PSPACE-complete in the combined input of the Kripke structure and formula.*

*Proof.* HyperLTL model checking on trees is PSPACE-complete in the combined input consisting of structure and formula [BF18]. Because HyperLTL is a subset of Fixpoint Hyper$^2$LTL$_{fp}$ it follows that MC[Fixpoint, Tree] is PSPACE-hard in the combined input. The containment in PSPACE follows from Lem. 22 and the fact that Fixpoint Hyper$^2$LTL$_{fp}$ is a subset of Hyper$^2$LTL. □

**Theorem 24.** *MC[Hyper$^2$LTL, Tree] is PSPACE-complete in the combined input of the Kripke structure and formula.*

*Proof.* MC[Hyper$^2$LTL$_{fp}$, Tree] is in PSPACE by Lem. 22. The rest of the proof follows directly from the fact that Hyper$^2$LTL is a superset of HyperLTL, for which model checking on trees is PSPACE-hard in the combined size of structure and formula [BF18]. □

# 6 Conclusion

We have analyzed the complexity of model checking Hyper$^2$LTL on acyclic structures. We have successfully shown matching upper and lower bounds for the model checking problem for Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ on tree-shaped and acyclic models. For full Hyper$^2$LTL we have shown matching upper and lower bounds for the model checking problem on tree-shaped models but only an upper bound for the model checking problem on acyclic models.

For Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ the complexity of the model checking problem depends polynomially on the number of traces contained in the model. Accordingly, MC[Fixpoint, Acyclic] is exponentially more complex than MC[Fixpoint, Tree], because acyclic models can contain exponentially more traces. Note that there is a similar exponential blow-up between Circuit Value which is as complex as MC[Fixpoint, Tree] [Pap94] and Succinct Circuit Value which is as complex as MC[Fixpoint, Acyclic].

In comparison to HyperLTL model checking, we can see that Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ model checking is not much more complex. On tree-shaped models, HyperLTL model checking is L-complete and Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ model checking is P-complete. Similarly, on acyclic models, HyperLTL model checking is PSPACE-complete and Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ model checking is EXP-complete.

Contrary to Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$, unrestricted Hyper$^2$LTL reasons over all sets of traces instead of only one set. This increases the time, but not the space needed for model checking because the required calculations for each set can be done one set after the other. Therefore, only one set per quantifier has to be present at once, exactly as in Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ model checking.

Another difference from Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$ model checking is that nondeterministically guessing quantifier instantiations is faster than calculating them deterministically. This comes from the fact that in Fixpoint Hyper$^2$LTL$_{\mathrm{fp}}$, there is a strategy given to compute the relevant set. For Hyper$^2$LTL we have not found such a strategy.

As a consequence of guessing instantiations, the complexity class changes depending on the number of second-order quantifier alternations in the given formula. This is because, for every second-order quantifier alternation, we have to change from guessing existentially quantified variables to guessing universally quantified variables or vice versa.

We have also shown that when considering the formula as well as the model as input, Hyper$^2$LTL model checking on acyclic models is as complex as HyperLTL model checking on acyclic models.

One aspect not covered in this thesis is a lower bound for MC[Hyper$^2$LTL, Acyclic]. We conjecture that MC[Hyper$^2$LTL, Acyclic] is $\Sigma^{\mathrm{EXP}}_{k+1}$-hard if the formula has $k$ second-order quantifier alternations and its outermost second-order quantifier is existential. Similarly, we conjecture that MC[Hyper$^2$LTL, Acyclic] is $\Pi^{\mathrm{EXP}}_{k+1}$-hard if the formula has $k$ second-order quantifier alternations and its outermost second-order quantifier is universal. This conjecture is based on the observation that for MC[Hyper$^2$LTL, Tree] as well as MC[Fixpoint, Acyclic] the model checking algorithms that simply enumerate all quantifier instantiations were asymptotically optimal. Further, one could compare the expressiveness of the Hyper$^2$LTL semantics that we use and the Hyper$^2$LTL semantics by Beutner et al. [Beu+23].

# Bibliography


[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity*. Cambridge: Cambridge University Press, 2009. URL: https://zbmath.org/?q=an:1193.68112.

[Bau+21] Jan Baumeister et al. "A Temporal Logic for Asynchronous Hyperproperties". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 694–717. ISBN: 978-3-030-81685-8.

[Beu+23] Raven Beutner et al. *Second-Order Hyperproperties*. 2023. arXiv: 2305.17935 [cs.LO].

[BF18] Borzoo Bonakdarpour and Bernd Finkbeiner. "The Complexity of Monitoring Hyperproperties". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. July 2018, pp. 162–174. DOI: 10.1109/CSF.2018.00019.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[BMP15] Laura Bozzelli, Bastien Maubert, and Sophie Pinchinat. "Unifying Hyper and Epistemic Temporal Logics". In: *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Andrew M. Pitts. Vol. 9034. Lecture Notes in Computer Science. Springer, 2015, pp. 167–182. DOI: 10.1007/978-3-662-46678-0_11. URL: https://doi.org/10.1007/978-3-662-46678-0%5C_11.

[Cla+14] Michael R. Clarkson et al. "Temporal Logics for Hyperproperties". In: *Principles of Security and Trust*. Ed. by Martín Abadi and Steve Kremer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 265–284. ISBN: 978-3-642-54792-8.

[Eis+03] Cindy Eisner et al. "Reasoning with Temporal Logic on Truncated Paths". In: *Computer Aided Verification*. Ed. by Warren A. Hunt and Fabio Somenzi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 27–39. ISBN: 978-3-540-45069-6.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractability*. A series of books in the mathematical sciences. New York, NY [u.a.]: Freeman, 1979. URL: http://www.ulb.tu-darmstadt.de/tocs/5617215X.pdf.

[Hsu+23] Tzu-Han Hsu et al. *Bounded Model Checking for Asynchronous Hyperproperties*. 2023. arXiv: 2301.07208 [cs.LO].

[KV99] Orna Kupferman and Moshe Y. Vardi. "Model Checking of Safety Properties". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*. Ed. by Nicolas Halbwachs and Doron A. Peled. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 172–183. DOI: 10.1007/3-540-48683-6_17. URL: https://doi.org/10.1007/3-540-48683-6%5C_17.

[MS99] Ron van der Meyden and Nikolay V Shilov. "Model checking knowledge and time in systems with perfect recall". In: *FSTTCS*. Vol. 1738. Springer. 1999, pp. 432–445.

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Theoretical computer science. Reading, Massachusetts: Addison-Wesley, [1994]. URL: http://www.gbv.de/dms/ilmenau/toc/12708035X.PDF.