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

Bachelor's Thesis

Complete Bounded Model Checking for Hyperproperties

Florian Bies
May 5, 2022

Advisors
Norine Coenen, M.Sc.
Niklas Metzger, M.Sc.

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Dr. Rayna Dimitrova

Statement

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.

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.

Place, Date	Signature

Abstract

Hyperproperties allow us to specify properties that relate multiple execution traces of systems with each other. The increased expressiveness comes at the cost of a significantly higher complexity of model checking. This gives rise to the development of feasible model checking algorithms that are applicable to particular fragments of hyperproperties.

In this context, recent work has generalized linear temporal logic (LTL) bounded model checking (BMC) to the hyperlogic HyperLTL, an extension of LTL to multiple traces. The fundamental idea of BMC is to establish a witness by considering only trace prefixes of bounded length. But the proposed algorithm is subject to the restriction that it is unable to argue about infinite traces such that formulas involving global requirements cannot be verified.

In this thesis, we attempt to eliminate this limitation. We discuss to what extent completeness techniques for LTL bounded model checking are applicable to HyperLTL. Aside from this, we develop novel approaches that are suitable to deal with trace quantification. Overall, we find ways to verify HyperLTL formulas incorporating an LTL invariant.

Acknowledgements

First of all, I am deeply grateful to Prof. Bernd Finkbeiner for his manifold support over the past year. His influence not only shaped my undergraduate studies, but extends far beyond into my future career.

I am also very thankful to Norine Coenen for having accompanied me on my journey towards this thesis from the very beginning. At the same time, I thank Niklas Metzger for being an excellent deputy for Norine and for staying with us. Both of them deserve great thanks for having spend countless hours in meetings with me.

Furthermore, I would like to thank Dr. Rayna Dimitrova for her willingness to reviewing this thesis.

Contents

1 Introduction **1**

2 Related Work **5**

3 Definitions **7**
3.1 Sequences 7
3.2 Systems 8
3.3 Automata 9
3.4 Hyperproperties 11
3.5 Temporal Logics 12
3.5.1 LTL 12
3.5.2 HyperLTL 15
3.6 Quantified Boolean Formulas 17

4 Bounded Model Checking **21**
4.1 LTL Bounded Model Checking 21
4.1.1 Encoding of the System 22
4.1.2 Encoding of the Formula 23
4.2 HyperLTL Bounded Model Checking 24

5 Completeness for LTL Bounded Model Checking **27**
5.1 Completeness Threshold 27
5.2 Loop Constraints 30

6 Completeness for HyperLTL Bounded Model Checking **33**
6.1 Alternation-free Fragment 33
6.2 Automaton-Based Completeness Threshold 35
6.3 Distinction from LTL Bounded Model Checking 39
6.4 $\exists\forall$ + LTL Invariant 40
6.4.1 Completeness Threshold 40
6.4.2 Incremental Algorithm 43
6.5 $\forall\exists$ + LTL Invariant 44

7 Evaluation **49**
7.1 Implementation . . . 49
7.1.1 BTOR2 . . . 50
7.1.2 Satisfiability Modulo Theories . . . 50
7.2 $\exists\forall$ + LTL Invariant . . . 51
7.2.1 System Representation . . . 52
7.2.2 Full Example . . . 54
7.2.3 More Benchmarks . . . 56
7.3 $\forall\exists$ + LTL Invariant . . . 62
7.4 Discussion . . . 66

8 Conclusion **67**
8.1 Summary . . . 67
8.2 Future Work . . . 68

Index **71**

Bibliography **73**

Chapter 1

Introduction

The concept of hyperproperties reasoning about sets of execution traces has been introduced by Clarkson and Schneider as a generalization of traditional trace properties [1]. Instead of being limited to the verification of individual system executions, they allow us to establish correctness statements with respect to multiple simultaneous runs. Hyperproperties have been proposed particularly as a means to express security properties that traditional non-hyper properties cannot [1]. The ability to relate multiple executions of a system enables us to for instance to specify various information-flow control properties [2]. However, potential applications of hyperlogics go far beyond this scope. In particular, many planning objectives may be considered as hyperproperties [3]. For instance, an optimal plan is characterized by the fact that it is at least as good as any other plan. This statement clearly relates two traces to each other.

Numerous temporal hyperlogics as well as associated model checking algorithms for hyperproperties have emerged since then [4]. Most notably, Clarkson et al. have proposed HyperLTL and HyperCTL* [5] as natural extensions of the corresponding standard temporal logics LTL [6] and CTL* [7], allowing us to express a wide range of hyperproperties in a unified logic. Verifying properties given as HyperLTL formulas is the main issue considered within the scope of this work.

The fundamental innovation of HyperLTL is the introduction of explicit trace quantification, enabling a refined reasoning about relations between multiple traces. While trace quantification in LTL is always universal implicitly, HyperLTL allows to intermix universal and existential trace quantification. As long as only the same sort of quantification appears in a HyperLTL formula, model checking may be reduced to the LTL case by composing all traces, or rather their underlying systems. Only the introduction of varied quantifiers enables us to formalise a richer set of properties.

Example 1.1 (Generalized noninterference)**.** The information-flow property *generalized noninterference (GNI)* [8] can be naturally expressed in HyperLTL. We define it with respect to a secret input h and a public output o of a system. GNI requires that for any two system executions π_1 and π_2, there is another execution π_3 that always agrees with π_1 on the secret input and with π_2 on the public output. This requirement prevents leakage of secret information. The existence of π_3 ensures that the public output is not affected by a change of the secret input. The corresponding HyperLTL formula is given below.

$$\forall \pi_1. \forall \pi_2. \exists \pi_3. \square((h_{\pi_1} \leftrightarrow h_{\pi_2}) \land (o_{\pi_1} \leftrightarrow o_{\pi_2}))$$ ⌟

The semantic and syntactical resemblance of LTL and HyperLTL suggests that well-known LTL model checking algorithms can possibly be lifted to HyperLTL. Indeed, many existing model checking procedures for HyperLTL stem from their LTL counterparts. For example, the first model checking algorithm for HyperLTL is inspired by automata-based LTL model checking [5]. This work already suggests that HyperLTL model checking algorithms are quite expensive in many cases. That particular construction involves the complementation of Büchi automata in order to handle quantifier alternations, causing an exponential blow-up in the overall algorithm. The overall insight is that model checking HyperLTL formulas including more than one quantifier sort appears to be often infeasible in practice if we attempt to apply such a general purpose algorithm without further information about the particular problem. This calls for more practical approaches which are expected to be efficient for at least some interesting fragments.

A symbolic LTL model checking algorithm that has proven to be very useful in practice is bounded model checking (BMC) [9]. What is bounded in this context is the length of the witnesses. In the case of LTL, these witnesses are simply trace prefixes. This approach is most successful if it suffices to consider short trace prefixes in order to establish a proof. Algorithmically, the bound is increased until a counterexample is found or even a proof can be established. The basic idea has already been successfully transferred to HyperLTL [10]. But the approach presented in this work is limited to properties that are refutable (or conversely verifiable) in finite time. For instance, a finite counterexample to the property $\forall \pi. \square a$ ('a holds globally') is a finite trace prefix leading to a state on which a does not hold. On the other hand, the property $\forall \pi. \diamond a$ ('a holds finally') can only be disproved by an infinite trace on which a never holds. In LTL bounded model checking, a counterexample in such a case is simply a looping path[1] in the corresponding system, identified by a finite path prefix containing some state twice. But it is not straightforward to apply this concept of looping paths to HyperLTL with quantifier alternation. Finding a single looping path is not sufficient in general since a HyperLTL formula constraints multiple traces jointly.

[1] A formal distinction between paths and traces appears in Section 3.2.

Within this work, we aim to overcome the aforementioned limitation imposed by the existing HyperLTL bounded model checking implementation. We intend to verify infinite requirements and thereby create a complete BMC procedure for HyperLTL. To this end, we first revisit state of the art LTL bounded model checking. Chapter 4 introduces the fundamental concepts of LTL BMC as well as its existing extension to HyperLTL. In the subsequent chapter, we show how completeness may be achieved for LTL BMC and discuss to what extend we can build upon techniques utilised there.

A common approach for achieving completeness in the context of BMC is to establish a completeness threshold. The underlying assumption is that any formula has a finite witness, i.e. it suffices to consider trace prefixes to determine its validity. A completeness threshold is an upper bound on the length of such trace prefixes, depending on both the system and the formula. How to obtain a general completeness threshold for HyperLTL is discussed in Section 6.2.

Since universal completeness thresholds for hyperproperties are often far too large to be usable in practice, we propose other algorithmic approaches that are capable of arguing about infinite computations in Sections 6.4 and 6.5. For this purpose, we exploit in particular the idea of considering loops occurring in paths which is also an essential concept in LTL BMC. In fact, we restrict ourselves to HyperLTL formulas with a once alternating quantifier prefix, followed by an LTL invariant. This may seem to be a severe restriction, but it allows us to focus on the actual extensions of existing work. Namely, we aim to verify infinite requirements imposed on differently quantified traces.

Beyond that, the thesis is structured as follows. In the next chapter, we relate our approach to other HyperLTL model checking procedures. Preliminary definitions including those of the relevant logics as well as our formal framework to specify systems are given in Chapter 3. LTL bounded model checking as well as the existing extension to HyperLTL is introduced in the subsequent chapter. After that, we discuss which concepts may lead to a complete bounded model checking procedure for LTL in Chapter 5, before proposing our own approaches for HyperLTL in Chapter 6. Those model checking algorithms are evaluated in Chapter 7. We conclude the thesis Chapter 8 and point out potential future work.

Chapter 2

Related Work

Several verification algorithms for fragments or extensions of HyperLTL have been developed so far [5, 11, 12, 13, 14, 10].

The first approach [5] is an adaptation of automaton-based LTL model checking [15]. The algorithm is applicable to HyperLTL with at most one quantifier alternation. Nevertheless, the complexity is exponential in the size of the system and even doubly exponential in the size of the formula which renders it infeasible in practice. By far the most expensive part is the complementation of a nondeterministic Büchi automaton [16] which is introduced in order to resolve the quantifier alternation.

Against the background of this huge complexity, further work has been done aiming at obtaining a more practical model checking procedure. Since HyperLTL model checking is PSPACE hard [5], the scope of feasible algorithms accomplishing this task is usually limited. Fortunately, there are several ways to simplify the problem without sacrificing too much of the additional expressiveness featured by hyperproperties that is relevant in practice.

First of all, if we abstain from quantifier alternation then model checking HyperLTL is just as complex as model checking LTL [11]. Finkbeiner et al. presented a practical model checking approach for the alternation-free fragment. The main insight is that it suffices to self-compose all quantified systems in this special case before standard LTL model checking algorithms can be applied [11]. Coenen et al. extended this construction such that up to one quantifier alternation is allowed, provided that a strategy for resolving the existential quantifiers can be found [14]. The latter two methods are implemented in the model checker MCHyper [17].

Bounded model checking for HyperLTL has been initially proposed by Hsu et al. [10], combined with the practical model checking tool HyperQube [18]. Essentially, it is a generalization of standard LTL bounded model checking [9, 19]. While the problem is reduced to the boolean satisfiability problem (SAT) [20] for LTL, it is reduced to quantified boolean satisfiability (QBF) [21] for HyperLTL. This has the benefit that the QBF solver takes care of resolving all quantifiers. Whereas the extension is quite simple and efficient in practice, it inherits the original bounded model checking characteristic of being inherently incomplete. In fact, it is even more limited than original LTL bounded model checking [22] since only finite path witnesses can be provided. Even seemingly simple HyperLTL formulas with an LTL invariant succeeding the quantifier prefix cannot be verified, which is the main motivation for our work. There are actually well known hyperproperties that can be brought into this shape (cf. Example 1.1).

Our work builds upon this bounded model checking approach for HyperLTL and aims to lift some of its limitations regarding completeness with techniques known from bounded LTL model checking. This is why we first revisit LTL BMC in chapter 4.

The performance of bounded model checking in the contexts mentioned above relies largely on the underlying satisfiability solver. Due to the success of the international SAT competition in the past two decades, numerous advanced and highly efficient SAT solvers have emerged until today [23]. This boosts the effectiveness of LTL BMC substantially. For HyperLTL BMC, we rely on a satisfiability solving procedure that is capable of dealing with quantification. Besides QBF, another promising candidate that fulfils this requirement is satisfiability modulo theories (SMT) which allows to augment booleans satisfiability with various logics and theories [24]. While SMT is far more widespread and dedicated solvers like Z3 [25] or cvc5 [26] are more mature than existing QBF tools, Hsu et al. observed that QBF solvers tend to outperform their SMT counterparts when applied to HyperLTL BMC [10]. This is presumably because SMT is far more extensive, such quantifier elimination is not the main issue. On the other hand, QBF solvers specifically focus on the problem of dealing with alternating quantifiers [27].

Recent work proposed to apply HyperLTL model checking on planning problems [3]. The motivation is that for instance shortest plans or robust plans may be characterized by their relation to all other plans. Thus, objectives like optimality or robustness can clearly be perceived as hyperproperties. While this approach is not competitive to state of the art tools [28] on classical planning problems like finding a shortest plan, hyperproperties are far more expressive. Inspired by this work, we discuss particularly problems from the planning domain to evaluate our algorithms.

Chapter 3

Definitions

In this chapter, we formally introduce the basic definitions and concepts we utilise throughout the thesis.

3.1 Sequences

A *sequence* is an enumerated listing of objects, denoted either in tuple-notation or simply as a concatenation of objects if the notation is unambiguous, e.g. (a, b, c) or abc. A *set* A is an unordered sequence of unique objects, stated in curly brackets. We define the set operations intersection $\cap$, union $\cup$ and complementation $\overline{A}$ (with respect to some contextual universe) as usual. For a family of n different sets $A_0, \ldots, A_{n-1}$, we define the *Cartesian product* $\times_{i=0}^{n-1} A_i \coloneqq \{(a_0, \ldots, a_{n-1}) \mid a_0 \in A_0, \ldots, a_{n-1} \in A_{n-1}\}$. If all sets are the same set A, we write A^n for the n-fold Cartesian product over A. The set of all infinite sequences with elements from A is denoted as A^ω. Furthermore, we define $\mathcal{P}(A) \coloneqq \{B \mid B \subseteq A\}$ as the *power set* of A.

Sequences consisting of natural numbers can be described succinct by the following conventions. For $i, j \in \mathbb{N}_0$ with $i \leq j$, we define $i..j \coloneqq (i, \ldots, j-1)$. If j should be included into the sequence, we write $i..{=}j \coloneqq (i, \ldots, j)$. If $i = 0$, we may omit i and if the sequence should be infinite, we may omit j.

The following definitions for finite sequences also apply to infinite sequences analogously. Let $a \in A^n$ be a sequence with $a = a_0 \ldots a_{n-1}$. We write $x \in a$ if there exists an index $i \in ..n$ such that $x = a_i$. Since sequences are ordered, we can access single elements by their position, so $a[i] \coloneqq a_i$ for $i \in ..n$. Similarly, we allow to extract subsequences by specifying a sequence of indices as an argument such that $a[(i_0, \ldots, i_{m-1})] \coloneqq (a[i_0], \ldots, a[i_{m-1}])$ where $i_j < n$ for $j \in ..m$.

Let $a_0, \ldots, a_{m-1} \in A^n$ be sequences. The *zip* operation maps m sequences to a single sequence whose elements are sequences of length m, formally $\mathit{zip}(a_0, \ldots, a_{m-1}) \coloneqq (a_0[0] \ldots a_{m-1}[0], \ldots, a_0[n-1] \ldots a_{m-1}[n-1])$.

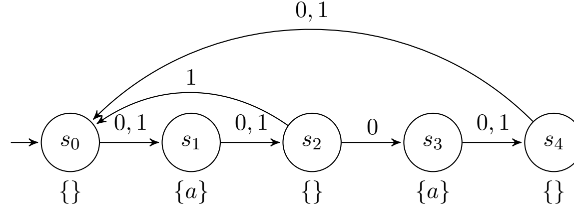

Figure 3.1: Graphical representation of a system with $S = \{s_0, s_1, s_2, s_3, s_4\}$, $Act = \{0, 1\}$ and $AP = \{a\}$. Each state is annotated with its label and each transition is annotated with its action(s). Hereinafter, we omit specifying the actions explicitly.

3.2 Systems

The systems we are reasoning about are a variation of transition systems. Most notably, we require that the transition function is total. This simplifies the notation a bit later and corresponds to the specification language for systems we consider in our experiments.

Definition 3.1 (System)**.** A *system* is a 6-tuple $K := (S, s_0, Act, \delta, AP, L)$, where

- S is the set of *states*,
- $s_0 \in S$ is the *initial state*,
- Act is the non-empty set of *actions*,
- $\delta \colon S \times Act \to S$ is the total *transition function*,
- AP is the set of *atomic propositions*,
- $L \colon S \to \mathcal{P}(AP)$ is the *labelling function*. ⌟

By requiring that δ is a total function, we ensure particularly that the transition system does not contain any terminal state, i.e. every state has a successor. A graphical representation of a sample system is given in Figure 3.1. We say that an atomic proposition $a \in AP$ holds in state $s \in S$ if $a \in L(s)$.

An *execution* $\alpha \in Act^\omega$ of a system is an infinite sequence of actions. We can associate a *path* $\sigma \in S^\omega$ to each execution, where

$$\begin{aligned} \sigma[0] &:= s_0 \\ \sigma[i+1] &:= \delta(\sigma[i], \alpha[i]) \qquad \text{for } i \in \mathbb{N}_0. \end{aligned}$$

We say that a path σ is *valid* with respect to a system K if there exists an execution α whose associated path is σ. Usually we only deal with valid paths, so if it is clear from the context we may omit stating explicitly that we are referring to a valid path.

A *trace* $t \in TR$ is a sequence of sets of atomic propositions, where $TR \coloneqq (\mathcal{P}(AP))^\omega$ is the set of all traces with respect to a set of atomic propositions AP. The trace of a path σ is the sequence of labels of the visited states, i.e. $Tr(\sigma) \coloneqq t$ where $t[i] = L(\sigma[i])$ for $i \in \mathbb{N}_0$. We say that a trace t is *valid* with respect to a system K if there exists a valid path σ with $Tr(\sigma) = t$. Thus, each valid trace can be associated with a (not necessarily unique) valid path. We often use this fact without further mention if it is convenient to reason about paths instead of traces. The set of all valid traces in a system K is $Traces(K)$. Again, by trace we usually mean valid trace.

If we talk about a *prefix* of an execution, a path or a trace the same definitions apply, except that we mean finite sequences. Variables for prefixes are usually annotated with a hat, e.g. $\hat{\sigma}$. Furthermore, we define a function δ^n on an execution prefix of size n as the n-fold application of δ.

$$\begin{aligned}
&\delta^n \colon Act^n \to S\\
&\delta^0(()) \coloneqq s_0\\
&\delta^{n+1}(\alpha) \coloneqq \delta(\delta^n(\alpha[..n]), \alpha[n]) \qquad \text{for } n \in \mathbb{N}_0
\end{aligned}$$

We define a successor function *Post* assigning each state its set of possible immediate successor states. The function is analogously defined on sets of states.

$$\begin{aligned}
&Post \colon S \to \mathcal{P}(S) && Post \colon \mathcal{P}(S) \to \mathcal{P}(S)\\
&Post(s) \coloneqq \bigcup_{\alpha \in Act} \{\delta(s, \alpha)\} && Post(Q) \coloneqq \bigcup_{s \in Q} Post(s)
\end{aligned}$$

While σ is a single path, $\varsigma \in (\mathcal{P}(S))^\omega$ may be perceived as the union of all valid paths along the time axis such that

$$\begin{aligned}
\varsigma[0] &\coloneqq \{s_0\}\\
\varsigma[i+1] &\coloneqq Post(\varsigma[i]) \qquad \text{for } i \in \mathbb{N}_0.
\end{aligned}$$

We say that a state is *reachable*, if it occurs in some valid path. Thus, the set of all reachable states is defined as

$$Reach(K) \coloneqq \bigcup_{i \in \mathbb{N}_0} \varsigma[i].$$

3.3 Automata

Next, we define Büchi automata as a common conception utilised in the context of model checking.

Definition 3.2 (Büchi automaton)**.** A *Büchi automaton* is a 5-tuple $\mathcal{A} \coloneqq (S, \Sigma, \delta, S_0, F)$, where

- S is the finite set of *states*,
- Σ is the *alphabet*,
- $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation*,
- $S_0 \subseteq S$ is the set of *initial states*,
- $F \subseteq S$ is the set of *accepting states*. ⌟

Let $w = w_0 \cdots \in \Sigma^\omega$ be a *word*. A *run* of $\mathcal{A}$ for w is a sequence $s_0 \cdots \in S^\omega$ such that $s_0 \in S_0$ and $(s_i, w_i, s_{i+1}) \in \Delta$ for $i \in \mathbb{N}_0$. A run is *accepting* if it contains infinitely many accepting states. Moreover, a word is accepted by the automaton if there exists an accepting run for it. The *language* of an automaton $\mathcal{L}(\mathcal{A})$ is defined as its set of accepted words. Büchi automata are closed under union $\cup$, intersection $\cap$ and complementation $\overline{\mathcal{A}}$.

A *safety* automaton $\mathcal{A}$ is a Büchi automaton whose language is a *safety property*. This means that for each $w \in \overline{\mathcal{L}(\mathcal{A})}$, there exists a *bad prefix* $\hat{w}$ of w such that $w' \in \overline{\mathcal{L}(\mathcal{A})}$ for all infinite extensions w' of $\hat{w}$. A *co-safety* automaton is a Büchi automaton whose complement is a safety automaton. An equivalent characterization is that for each $w \in \mathcal{L}(\mathcal{A})$, there exists a *good prefix* $\hat{w}$ of w such that $w' \in \mathcal{L}(\mathcal{A})$ for all infinite extensions w' of $\hat{w}$. Safety automata are closed under intersection and co-safety automata are closed under union. In terms of complementation, they are dual.

Each system $K = (S, s_0, Act, \delta, AP, L)$ can be associated with a safety automaton $\mathcal{A}_K \coloneqq (S, \Sigma, \Delta, \{s_0\}, S)$ where $\Sigma = \mathcal{P}(AP)$ and

$$(s, l, s') \in \Delta \iff (s' = \delta(s, \alpha) \wedge l = L(s) \text{ for some } \alpha \in Act).$$

Note that valid traces TR of K and accepted words of $\mathcal{A}_K$ coincide, i.e. $TR = \mathcal{L}(\mathcal{A}_K)$.

Assume that $\Sigma = \mathcal{P}(AP)$ and let $A \subseteq AP$. For a word $w \in \Sigma^\omega$, we define the *projection* onto A as $w|_A \in (\mathcal{P}(A))^\omega$ such that for each $i \in \mathbb{N}_0$, $w|_A[i] = w[i] \setminus \overline{A}$. Furthermore, we define the projection $\mathcal{A}|_A$ on automata, such that $\mathcal{L}(\mathcal{A}|_A) = \{w|_A \mid w \in \mathcal{L}(\mathcal{A})\}$. The alphabet of $\mathcal{A}|_A$ is $\mathcal{P}(A)$ and the transition function Δ' for $\mathcal{A}|_A$ is defined such that it satisfies $(s, l, s') \in \Delta' \iff s' \in \{\delta(s, l \cup l') \mid l' \in \mathcal{P}(\overline{A})\}$. Other than that, the definition coincides with $\mathcal{A}$. Note that projection preserves the (co-)safety property of automata.

3.4 Hyperproperties

If we talk about properties in this work, we are usually referring to properties reasoning about traces. A standard (non-hyper) *trace property* $P \in \mathcal{P}(TR)$ describes a set of traces. A trace property always talks about characteristics of a single trace. In order to determine whether a system satisfies a trace property P, we have to check whether the trace of each system execution is contained in P.

Example 3.1. Consider again the system given in Figure 3.1 and the following two properties, where a is an atomic proposition:

1. a must not hold at two subsequent points in time on one trace, i.e. for one trace t, there is no index $i \in \mathbb{N}_0$ such that a holds in both states $t[i]$ and $t[i+1]$.

2. a must not hold at two subsequent points in time on any traces, i.e. for any traces t, t', there is no index $i \in \mathbb{N}_0$ such that a holds in both states $t[i]$ and $t'[i+1]$.

The first property is clearly a common trace property since it only talks about one trace. It holds since any state labelled with a is always followed by some state that is not labelled with any atomic proposition. The second property requires in addition that if a holds at position i of some trace, then a must not hold at position $i+1$ of any other trace. This cannot be expressed as an ordinary trace property any more since it cannot be verified by considering all traces individually. Still, we would like to be able to verify automatically that the second property does not hold. ⌟

This is where *hyperproperties* [1] come into play. They enable us to specify how different system traces relate to each other. Formally, a hyperproperty is a property $H \in \mathcal{P}(\mathcal{P}(TR))$, i.e. a set of trace properties. A set of traces T satisfies H if and only if it is contained in H.

Example 3.1 (Continued)**.** Consider a formal specification of the two properties stated above.

1. $\forall t \in \mathit{Traces}(K).\forall i \in \mathbb{N}_0.\neg(a \in L(t[i]) \wedge a \in L(t[i+1]))$

2. $\forall t, t' \in \mathit{Traces}(K).\forall i \in \mathbb{N}_0.\neg(a \in L(t[i]) \wedge a \in L(t'[i+1]))$

It is evident that the former property talks about individual traces while the latter (hyper)property talks about tuples of traces. The second property is indeed violated by the trace prefixes $\hat{t} := \emptyset\{a\}\emptyset\{a\}\emptyset$ and $\hat{t}' := \emptyset\{a\}\emptyset\emptyset\{a\}$, obtained from the path prefixes $\hat{\sigma} := s_0 s_1 s_2 s_3 s_4 \ldots$ and $\hat{\sigma}' := s_0 s_1 s_2 s_0 s_1 \ldots$ respectively. ⌟

3.5 Temporal Logics

We use *temporal logics* in order to reason about systems. Temporal logics give us a formal framework to specify trace properties and hyperproperties. In this work, we consider only the linear-time spectrum of temporal logics logics. Our notion of time is both linear and discrete.

In the following, we introduce the temporal logic LTL as well as HyperLTL, its extension to hyperproperties.

3.5.1 LTL

Linear temporal logic (LTL) [6] is a well-known specification language for linear-time properties.

Definition 3.3 (LTL Syntax)**.** An LTL formula φ is generated by the grammar

$$\varphi ::= \mathit{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi,$$

where $a \in AP$ is an *atomic proposition.* ⌟

We say that $\bigcirc$ represents the *next* operator and $\mathcal{U}$ denotes the *until* operator. Besides *negation* $\neg$ and *conjunction* $\wedge$, we allow other conventional propositional logic operators like *disjunction* $\vee$, *implication* $\rightarrow$ and *equivalence* $\leftrightarrow$ as syntactic sugar with their usual meanings. We define the semantics as a satisfaction relation on traces.

Definition 3.4 (LTL Semantics)**.** Let t be a trace and let $\varphi, \varphi_1, \varphi_2$ be LTL formulas.

$$\begin{aligned}
&t \models \mathit{true} && \\
&t \models a && \iff a \in t[0] \\
&t \models \neg\varphi && \iff t \not\models \varphi \\
&t \models \varphi_1 \wedge \varphi_2 && \iff t \models \varphi_1 \wedge t \models \varphi_2 \\
&t \models \bigcirc\varphi && \iff t[1..] \models \varphi \\
&t \models \varphi_1\,\mathcal{U}\,\varphi_2 && \iff \exists i \in \mathbb{N}_0\colon\ (t[i..] \models \varphi_2 \wedge \forall j \in ..i\colon t[j..] \models \varphi_1)
\end{aligned}$$
⌟

The temporal connective $\bigcirc\varphi$ requires that φ holds on the next state of trace t. Intuitively, the until expresses that the right sub-formula must hold at some point in the future and the left sub-formula must hold permanently beforehand. Based on the until, we define two more derived temporal operators. $\Diamond\varphi \coloneqq \mathit{true}\,\mathcal{U}\,\varphi$ demands that φ must hold at some point in time and $\square\varphi \coloneqq \neg\Diamond\neg\varphi$ requires that φ holds on the whole trace. A formula of the form $\square\psi$ is called an *invariant*, where ψ is a *propositional formula*, i.e. an LTL formula without any temporal operators.

Example 3.2. The first trace property given in Example 3.1 may be written as an LTL property: $\Box(\neg(a \land \bigcirc a))$ ⌟

The *language* $\mathcal{L}(\varphi)$ of a formula φ is the set of traces that satisfy φ. We say that a formula φ is *valid* or *satisfied* for a system K, written $K \models \varphi$, if $t \models \varphi$ for each trace $t \in \mathit{Traces}(K)$.

Sometimes it is useful to assume that negations may only occur immediately in front of atomic propositions. Formulas in this shape are said to be in *negation normal form* (NNF). In order to obtain a linear translation from general LTL formulas into NNF, we introduce the *release* operator defined as $\varphi_1 \mathcal{R} \varphi_2 := \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$. Intuitively, it expresses that φ_2 must hold the whole time, except if φ_1 was true at some previous point in time. Or rephrased, φ_1 releases $\Box\varphi_2$.

Definition 3.5 (LTL Negation Normal Form)**.** An LTL formula φ is in negation normal form if it can be derived from the grammar

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid a \mid \neg a \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi \mid \varphi \mathcal{R} \varphi.$$ ⌟

Until and release satisfy an expansion law respectively which can be deduced from the semantics, namely $\varphi_1 \mathcal{U} \varphi_2 = \varphi_2 \lor (\varphi_1 \land \bigcirc(\varphi_1 \mathcal{U} \varphi_2))$ and $\varphi_1 \mathcal{R} \varphi_2 = \varphi_2 \land (\varphi_1 \lor \bigcirc(\varphi_1 \mathcal{R} \varphi_2))$.

Bounded Semantics

In the context of bounded model checking, it is often the case that we need to consider a bounded version of the LTL semantics which allows us to reason about finite trace prefixes [9]. The idea is to come up with a modified relation $\models^k$ which essentially constrains the usual definition to trace prefixes of size k. Most importantly, it fulfils the following characteristic.

Proposition 3.1.

$$K \models^k \varphi \implies K \models \varphi.$$ ⌟

In some sense, $\models^k$ is an under-approximation of the unbounded semantics. All formulas that are satisfied in this bounded semantics do also hold in the regular semantics. Thus, we refer to it as the *pessimistic semantics*, following the designation in [10]. Note that $\models^k$ is notation for $\models^k_0$ as defined below.

Definition 3.6 (Pessimistic LTL semantics)**.** Let $k \in \mathbb{N}_0$ be the bound and let $k' \in ..{=}k$. Let t be a trace (prefix) of length greater than k and let $\varphi, \varphi_1, \varphi_2$ be LTL formulas in NNF.

$$
\begin{aligned}
& t \models^k_{k'} \mathit{true} \\
& t \not\models^k_{k'} \mathit{false} \\
& t \models^k_{k'} a && \iff a \in t[k'] \\
& t \models^k_{k'} \neg a && \iff a \notin t[k'] \\
& t \models^k_{k'} \varphi_1 \wedge \varphi_2 && \iff t \models^k_{k'} \varphi_1 \wedge t \models^k_{k'} \varphi_2 \\
& t \models^k_{k'} \varphi_1 \vee \varphi_2 && \iff t \models^k_{k'} \varphi_1 \vee t \models^k_{k'} \varphi_2 \\
& t \models^k_{k'} \bigcirc \varphi && \iff k' < k \wedge t \models^k_{k'+1} \varphi \\
& t \models^k_{k'} \varphi_1 \,\mathcal{U}\, \varphi_2 && \iff \exists i \in k'..{=}k \colon \left(t \models^k_i \varphi_2 \wedge \forall j \in k'..i \colon t \models^k_j \varphi_1\right) \\
& t \models^k_{k'} \varphi_1 \,\mathcal{R}\, \varphi_2 && \iff \exists i \in k'..{=}k \colon \left(t \models^k_i \varphi_1 \wedge \forall j \in k'..{=}i \colon t \models^k_j \varphi_2\right)
\end{aligned}
$$

⌟

We assume that the semantics always evaluates to false if $k' = k$ in the next-case. The definition of the release-case implies that $\square \psi$ always evaluates to false in the pessimistic semantics. The correctness of Proposition 3.1 for this definition is an immediate consequence of lemma 1 stated in [22].

Similarly, we can come up with yet another modified relation $\dot{\models}^k$ that provides an overapproximation to the standard semantics. Thus, we refer to it as the *optimistic semantics*, again following the designation in [10]. It can be obtained by dropping all future requirements, i.e. those exceeding the observable horizon of a trace prefix bounded by k.

Definition 3.7 (Optimistic LTL semantics)**.** Let $k \in \mathbb{N}_0$ be the bound and let $k' \in ..{=}k$. Let t be a trace (prefix) of length greater than k and let $\varphi, \varphi_1, \varphi_2$ be LTL formulas in NNF.

$$
\begin{aligned}
& t \,\dot{\models}^k_{k'} \bigcirc \varphi && \iff k' \geq k \vee t \,\dot{\models}^k_{k'+1} \varphi \\
& t \,\dot{\models}^k_{k'} \varphi_1 \,\mathcal{U}\, \varphi_2 && \iff \left(t \models^k_{k'} \varphi_1 \,\mathcal{U}\, \varphi_2\right) \vee \left(\forall i \in k'..{=}k \colon t \,\dot{\models}^k_i \varphi_1\right) \\
& t \,\dot{\models}^k_{k'} \varphi_1 \,\mathcal{R}\, \varphi_2 && \iff \left(t \models^k_{k'} \varphi_1 \,\mathcal{R}\, \varphi_2\right) \vee \left(\forall i \in k'..{=}k \colon t \,\dot{\models}^k_i \varphi_2\right)
\end{aligned}
$$

The remaining rules are equivalent to those given in Definition 3.6, modulo replacing $\models$ by $\dot{\models}$. ⌟

For this relation, the reverse direction of Proposition 3.1 holds.

Proposition 3.2.

$$K \models \varphi \implies K \,\dot{\models}^k \varphi$$

⌟

3.5.2 HyperLTL

HyperLTL is a formal logic that enables us to express a wide range of hyperproperties conveniently. It was proposed by Clarkson et al. [5] as a natural generalization of LTL.

The essential innovation of HyperLTL is the ability to simultaneously quantify over multiple system executions. In order to accomplish this, the syntax of LTL is extended with universal and existential trace quantifiers, each followed by a trace variable respectively. In general, any HyperLTL formula is just an LTL formula with a quantifier prefix, except that each atomic proposition is now explicitly associated with a trace variable π which is carried as an index.

Definition 3.8 (HyperLTL Syntax)**.** A HyperLTL formula Φ is generated by the grammar.

$$\Phi ::= \exists\pi.\Phi \mid \forall\pi.\Phi \mid \varphi$$
$$\varphi ::= a_\pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi,$$

where $a \in AP$ is an *atomic proposition* and $\pi \in \mathcal{V}$ is a *trace variable*. ⌟

The inner part of a HyperLTL formula is basically an LTL formula, thus it may contain all of the syntactic sugar mentioned above. Furthermore, we say that a HyperLTL formula is in negation normal form if its LTL body (φ in the above definition) is in this form.

We identify certain syntactical fragments of HyperLTL by the shape of their quantifier prefix. For a formula $\Phi := Q_1\pi_1.\ldots.Q_n\pi_n.\varphi$ in the notation of Definition 3.8 (φ is quantifier free), the quantifier prefix is represented as the corresponding sequence of quantifiers $Q_1 \ldots Q_n \in \{\forall, \exists\}^n$. Φ contains a *quantifier alternation* each time an universal (existential) quantifier is immediately followed by an existential (universal) quantifier in the quantifier prefix. The *alternation depth* of a HyperLTL formula is the number of quantifier alternations occurring in its quantifier prefix.

We allow each trace variable to quantify over the traces from another system. Therefore, we define the semantics of a HyperLTL formula φ with respect to a *system environment* Γ that maps identifiers κ to concrete systems.

The validity of a formula φ with respect to a system environment Γ and a *trace assignment* Π mapping trace variables to traces is written as $\Pi \models_\Gamma \varphi$. The empty trace assignment is denoted by $\emptyset$. An update of a trace assignment Π is defined by $\Pi[\pi \mapsto t]$ where $\Pi[\pi \mapsto t](\pi') := \text{if}\, \pi = \pi' \,\text{then}\, t \,\text{else}\, \Pi(\pi')$. Additionally, $\Pi[X]$ is a notation for the function mapping a path variable π to $\Pi(\pi)[X]$ for a sequence of natural numbers X. We say that a formula φ is *valid* or *satisfied* in a given environment Γ if $\emptyset \models_\Gamma \varphi$. This satisfaction relation is defined analogously to the respective relation for LTL.

Definition 3.9 (HyperLTL Semantics)**.** Let Π be a trace assignment, let Γ be a system environment, let $\varphi, \varphi_1, \varphi_2$ be LTL formulas and let Φ be a HyperLTL formula.

$$
\begin{aligned}
\Pi &\models_\Gamma a_\pi && \iff a \in \Pi(\pi)[0] \\
\Pi &\models_\Gamma \neg\varphi && \iff \Pi \not\models_\Gamma \varphi \\
\Pi &\models_\Gamma \varphi_1 \wedge \varphi_2 && \iff \Pi \models_\Gamma \varphi_1 \wedge \Pi \models_\Gamma \varphi_2 \\
\Pi &\models_\Gamma \bigcirc\varphi && \iff \Pi[1..] \models_\Gamma \varphi \\
\Pi &\models_\Gamma \varphi_1 \mathcal{U} \varphi_2 && \iff \exists i \in \mathbb{N}_0 \colon (\Pi[i..] \models_\Gamma \varphi_2 \wedge \forall j \in ..i \colon \Pi[j..] \models_\Gamma \varphi_1) \\
\Pi &\models_\Gamma \exists_\kappa \pi.\Phi && \iff \exists t \in \mathit{Traces}(\Gamma(\kappa)) \colon \Pi[\pi \mapsto t] \models_\Gamma \Phi \\
\Pi &\models_\Gamma \forall_\kappa \pi.\Phi && \iff \forall t \in \mathit{Traces}(\Gamma(\kappa)) \colon \Pi[\pi \mapsto t] \models_\Gamma \Phi
\end{aligned}
$$
⌟

Usually, we do not specify Γ and κ explicitly. Instead, we enumerate all n trace quantifiers occurring in the formula and assume that each quantifier refers to its own system. Now it suffices to specify a sequence $K \coloneqq (K_1, \ldots, K_n)$ of systems such that $\Gamma(i) \coloneqq K_i$ for $i \in 1..{=}n$. Validity is denoted as $K \models \varphi$ in this case. If the alphabets of all systems are pairwise disjoint, we may omit the trace variable indices of atomic propositions. Moreover, if we mention a system-related object like a path σ_i annotated with an index i, this object belongs to system K_i. A sequence $s \coloneqq (s_1, \ldots, s_n)$ where each s_i is a state of K_i for $i \in 1..{=}n$ is also named a *state* in this context.

Sometimes, we allow negations to appear in the quantifier prefix. The negation of a HyperLTL formula can be obtained by applying the equation $\neg\exists\pi.\varphi = \forall\pi.\neg\varphi$ multiple times. This leads to the following property that holds for any HyperLTL formula φ.

Proposition 3.3.

$$K \models \varphi \iff K \not\models \neg\varphi$$
⌟

Note that this equivalence does not hold for LTL since all LTL formulas are implicitly universally quantified. A system containing the traces $\{\}^\omega$ and $\{\}\{a\}^\omega$ fulfils neither $\Diamond a$ nor its negation $\Box\neg a$. An LTL formula φ is equisatisfiable to the HyperLTL formula $\forall\pi.\varphi$, so HyperLTL is indeed a generalization of LTL. Negating an LTL formula does not change the (implicit) quantification.

Example 3.3 (Language Containment)**.** We can specify *language containment* as a HyperLTL property. Given two systems K_1, K_2 with a common alphabet AP, this property requires that the traces of a system K_1 constitute a subset of the traces of another system K_2.

$$\forall\pi_1.\exists\pi_2.\Box\left(\bigwedge_{a \in AP} a_{\pi_1} \leftrightarrow a_{\pi_2}\right)$$
⌟

Bounded Semantics

Unsurprisingly, we may define bounded versions of the HyperLTL semantics as well. The definitions are completely analogously to the LTL case, which is why we omit them here. We use the same notation. Of course, the bounded semantics have the same characteristics stated in the LTL case. Both are also stated in [10] as the first two cases of Lemma 2.

Proposition 3.4.

$$K \models^k \Phi \implies K \models \Phi$$ ⌟

Proposition 3.5.

$$K \models \Phi \implies K \dot{\models}^k \Phi$$ ⌟

The following proposition gives a little more refined overview over the relation between the different semantics.

Proposition 3.6. *Let $k, k' \in \mathbb{N}_0$ with $k' \leq k$.*

$$K \models^{k'} \Phi \implies K \models^k \Phi \implies K \models \Phi \implies K \dot{\models}^k \Phi \implies K \dot{\models}^{k'} \Phi$$ ⌟

3.6 Quantified Boolean Formulas

Quantified boolean formulas are propositional logic formulas extended with explicit quantification over variables. The syntax is given by the following grammar, plus the usual syntactic sugar for the boolean connectives $0, \vee, \Rightarrow, \Leftrightarrow$ and 'if then else'. Note that the notation of the arrows and constants differs from LTL such that it is easier to distinguish them visually. Moreover, the arrows differ from the longer logical arrows.

Definition 3.10 (Quantified boolean formula syntax)**.** A *Quantified boolean formula* Ψ is generated by the following grammar.

$$\Psi ::= \exists x.\Psi \mid \forall x.\Psi \mid \psi$$
$$\psi ::= 1 \mid x \mid \neg\psi \mid \psi \wedge \psi \mid \Psi$$ ⌟

Example 3.4. The quantified boolean formula

$$\forall x.\exists y.(x \Rightarrow y) \Rightarrow \neg y$$

expresses that no matter which boolean value x takes, it is possible to chose a boolean value for y (which may depend on x) such that the inner propositional formula is satisfied. ⌟

A quantified boolean formula is in *prenex normal form* if it can be derived from the grammar without using the last rule in the definition of ψ. Thus, a formula in prenex normal form consists of a quantifier prefix, followed by a propositional formula. Most formulas we consider hereinafter lie in this restricted fragment.

We say that $Q \in \{\forall, \exists\}$ is a *quantifier*. Each *variable* x is taken from a set of variables X. $x \in X$ is a *free* variable in the boolean formula ψ if $\psi = x$ or if x is free in all sub-formulas of ψ. A quantified boolean formula of the form $Qx.\psi$ binds each occurrence of x as a free variable in ψ such that x is not free in $Qx.\psi$. A quantified boolean formula is *closed* if it does not contain any free variables.

A *variable assignment* $v\colon X \to \mathbb{B}$ maps variables to truth values from the boolean domain $\mathbb{B} \coloneqq \{0, 1\}$. The empty assignment is denoted as $\emptyset$. An update of a variable assignment v is written as $v[x \mapsto b]$ where $v[x \mapsto b](x') \coloneqq \text{if}\, x = x' \,\text{then}\, b \,\text{else}\, v(x')$ for $b \in \mathbb{B}$.

The satisfiability problem for quantified boolean formulas (QBF) is defined by a relation $\models$.

Definition 3.11 (QBF semantics)**.** Let v be a variable assignment and let ψ, ψ_1, ψ_2 be quantified boolean formulas.

$$
\begin{aligned}
&v \models 1 \\
&v \models \neg\psi && \iff v \not\models \psi \\
&v \models \psi_1 \wedge \psi_2 && \iff (v \models \psi_1) \wedge (v \models \psi_2) \\
&v \models \exists x.\psi && \iff (v[x \mapsto 1] \models \psi) \vee (v[x \mapsto 0] \models \psi) \\
&v \models \forall x.\psi && \iff (v[x \mapsto 1] \models \psi) \wedge (v[x \mapsto 0] \models \psi)
\end{aligned}
$$
⌟

A closed quantified boolean formula ψ is *satisfied* if and only if $\emptyset \models \psi$. In this case, we write $\mathrm{SAT}(\psi)$. In the opposite case, we write $\mathrm{UNSAT}(\psi)$.

Example 3.4 (Continued)**.** The quantified boolean formula given before is satisfied. We can convince ourselves that this is indeed true by applying the rules provided by the semantics. Another way to think about this is that if we chose y to be $\neg x$, then the inner propositional formula holds as witnessed by well-known logical equalities. ⌟

We allow some more syntactic sugar for quantified boolean formulas. If $\mathbf{x} \in X^n$ is a sequence of variables, then we write $Q\mathbf{x}.\psi$ as a shortcut for $Q\mathbf{x}[0]. \ldots Q\mathbf{x}[n-1].\psi$. Furthermore, we allow to apply boolean connectives on variables sequences $\mathbf{x}, \mathbf{x}' \in X^n$ of equal length. The respective operations are applied element-wise. For example, $\mathbf{x} \wedge \mathbf{x}'$ stands for the conjunction $(\mathbf{x}[0] \wedge \mathbf{x}'[0]) \wedge \cdots \wedge (\mathbf{x}[n-1] \wedge \mathbf{x}'[n-1])$. In a similar manner, we define equality $=$ and inequality $\neq$ for sequences of variables.

Finally, we define *boolean formulas* as quantified boolean formulas that do not contain any universal quantifier. In this case, we may omit the explicit quantification. The satisfiability problem for boolean formulas is known as the SAT problem.

Satisfiability modulo theories (SMT) generalizes boolean satisfiability even further than QBF. For instance, it allows to reason about bit-vectors or even real numbers and supports arithmetic operations. SMT extends boolean formulas with new logics as needed. Note that the fraction of SMT that is relevant in our context corresponds mostly to QBF such that we do not need to introduce a new formalism for SMT. In practice, using the bit-vector logic of SMT can be convenient, depending on how the system is encoded. A genuine extension of SMT that we need at some point are uninterpreted functions such that we may quantify over variables representing functions.

Chapter 4

Bounded Model Checking

Bounded model checking (BMC) is a widespread symbolic model checking technique that looks for witnesses bounded in length. If those can be identified easily, BMC is a very promising model checking approach. The fundamental approach we discuss here was initially proposed by Biere et al. in 1999 [9]. Its effectiveness relies particularly on the concise symbolic representation of the state space [29] as well as the capabilities of modern SAT solvers. We start by introducing BMC for LTL before presenting the BMC algorithm for HyperLTL derived thereof. Note that the LTL model checking algorithm presented in this chapter omits one essential feature introduced in [9], namely utilising looping witnesses. We moved it to the next chapter since it is no prerequisite of current state of the art HyperLTL BMC.

4.1 LTL Bounded Model Checking

Bounded model checking for LTL is particularly well suited to find violations of a property. Intuitively, this is because it suffices to provide a single trace in order to disprove an LTL formula. A witness is also called *counterexample* in this case, it is a trace that fulfils the negated property. Since the length of a witness is bounded in BMC, a counterexample is a finite execution prefix of size k. It represents a proof for $K \not\models \varphi$ given a system K and a property φ. Algorithmically, the *bound* k is increased stepwise until a counterexample of the respective size occurs. In this basic form, we obtain only a semi-decision procedure which means that if the property holds and thereby no counterexample exists, the algorithm does not terminate. We outline how this procedure can be applied to LTL properties.

In the first step, the algorithm constructs a boolean formula that is satisfiable if and only if a counterexample of length k (the current bound) exists. For now, a counterexample is an execution of size k such that the corresponding trace prefix $\hat{t}$ cannot be extended to a valid trace that satisfies the formula φ. We may formalize this in terms of the optimistic semantics as $\hat{t} \not\models^{k} \varphi$. The existence of such a trace violates the requirement that all traces must satisfy φ, thus $\hat{t}$ is a witness for $K \not\models \varphi$. An equivalent characterization is that all extensions of $\hat{t}$ to a valid trace satisfy the negated formula, i.e. $\hat{t} \models^{k} \neg\varphi$, which means that $\hat{t}$ violates the formula in any case.

Based on the second characterization, we define an encoding $[\![\neg\varphi]\!]^k$ describing a boolean formula whose proof of satisfiability immediately corresponds to a counterexample. What remains to be discussed is how to unfold both the system and the formula up to bound k in order to obtain such a boolean formula.

4.1.1 Encoding of the System

A formula is always evaluated based on the system states. Those in turn are solely dependent on the respective execution prefix. Thus, an execution $\alpha \in Act^k$ constitutes the set of variables of the boolean formula. Now the system state at step k', where $k' \leq k$, is defined by $\delta^{k'}(\alpha[..k'])$. Usually we assume that σ is the path obtained from α such that $\sigma[k'] = \delta^{k'}(\alpha[..k'])$.

In order to encode the system into a boolean formula representation, we just have to encode all states and actions as bit-vectors, i.e. sequences of boolean values. Furthermore, we define one separate labelling function per atomic proposition. The construction is straightforward. Therefore we do not formalize it in full detail, but instead illustrate it at an example.

Example 4.1. Consider again the system depicted in Figure 3.1. This system can be encoded into boolean formulas as written below.

$$
\begin{aligned}
S :=& \{s_0, s_1, s_2, s_3, s_4\} = \{000, 001, 010, 011, 100\} \\
Act :=& \{0, 1\} \\
\delta(s, \alpha) :=& \,\mathrm{if}\, s \Leftrightarrow 000 \,\mathrm{then}\, 001 \,\mathrm{else} \\
& \mathrm{if}\, s \Leftrightarrow 001 \,\mathrm{then}\, 010 \,\mathrm{else} \\
& \mathrm{if}\, s \Leftrightarrow 010 \,\mathrm{then}(\mathrm{if}\, \alpha \,\mathrm{then}\, 000 \,\mathrm{else}\, 011) \,\mathrm{else} \\
& \mathrm{if}\, s \Leftrightarrow 011 \,\mathrm{then}\, 100 \,\mathrm{else}\, 000 \\
\mathcal{L}_a :=& \,\mathrm{if}(s \Leftrightarrow 001) \vee (s \Leftrightarrow 011) \,\mathrm{then}\, 1 \,\mathrm{else}\, 0
\end{aligned}
$$

We can now check for example whether the execution 010 leads to a state where a holds by computing

$$
\delta^3(010) = \delta(\delta(\delta(000, 0), 1), 0) = \delta(\delta(001, 1), 0) = \delta(010, 0) = 011 \text{ and}
$$
$$
\mathcal{L}_a(\delta^3(010)) = \mathcal{L}_a(011) = 1.
$$
⌟

4.1.2 Encoding of the Formula

Given the system encoding described in the previous section, we would like to construct a formula encoding $[\![\neg\varphi]\!]^k$ that is satisfiable if and only if a counterexample of length k exists. Assuming that $\neg\varphi$ formula is in NNF, the most basic translation looks as follows. Note that this encoding does not recognize all counterexamples yet.

$:=$	$k' \leq k$	$k' > k$
$[\![true]\!]^k_{k'}$	1	0
$[\![false]\!]^k_{k'}$	0	0
$[\![a]\!]^k_{k'}$	$\mathcal{L}_a(\sigma[k'])$	0
$[\![\neg a]\!]^k_{k'}$	$\neg\mathcal{L}_a(\sigma[k'])$	0
$[\![\varphi_1 \wedge \varphi_2]\!]^k_{k'}$	$[\![\varphi_1]\!]^k_{k'} \wedge [\![\varphi_2]\!]^k_{k'}$	0
$[\![\varphi_1 \vee \varphi_2]\!]^k_{k'}$	$[\![\varphi_1]\!]^k_{k'} \vee [\![\varphi_2]\!]^k_{k'}$	0
$[\![\bigcirc\varphi]\!]^k_{k'}$	$[\![\varphi]\!]^k_{k'+1}$	0
$[\![\varphi_1 \,\mathcal{U}\, \varphi_2]\!]^k_{k'}$	$[\![\varphi_2]\!]^k_{k'} \vee ([\![\varphi_1]\!]^k_{k'} \wedge [\![\varphi_1 \,\mathcal{U}\, \varphi_2]\!]^k_{k'+1})$	0
$[\![\varphi_1 \,\mathcal{R}\, \varphi_2]\!]^k_{k'}$	$[\![\varphi_2]\!]^k_{k'} \wedge ([\![\varphi_1]\!]^k_{k'} \vee [\![\varphi_1 \,\mathcal{R}\, \varphi_2]\!]^k_{k'+1})$	0

We define $[\![\varphi]\!]^k := [\![\varphi]\!]^k_0$. The variable k' in the lower index always refers to the unrolling depth at which the formula is evaluated. The system K is always treated as an implicit argument of the encoding. In order to determine whether an atomic proposition holds at a certain point in time, we need to access the trace of the system as defined in the previous section. This is the only point where both encodings interact with each other.

The formula encoding is quite close to the (bounded) semantics of LTL and utilises the expansion law for until and release. This leads to the following observation.

Lemma 4.1. *Let K be a system, let φ be an LTL formula and let $k \in \mathbb{N}$.*

$$\mathrm{SAT}\left([\![\varphi]\!]^k\right) \iff \exists t \in \mathit{Traces}(K) \colon t \models^k \varphi$$

Proof. The more general statement $\mathrm{SAT}\left([\![\varphi]\!]^k_{k'}\right) \iff \exists t \in \mathit{Traces}(K) \colon t \models^k_{k'} \varphi$ for $k' \leq k$ can be proved by straightforward structural induction on φ with k' quantified. In the until/release case, we may use the expansion law. $\square$

Apparently, the encoding coincides with the pessimistic semantics of LTL. Note that if we replace the zeros in the right column of the encoding above by ones, we obtain an encoding $[\![\dot{\varphi}]\!]^k$ for the optimistic semantics in the same manner.

Lemma 4.2. *Let K be a system, let φ be an LTL formula and let $k \in \mathbb{N}$.*

$$\mathrm{SAT}\left(\llbracket \dot{\varphi} \rrbracket^k\right) \iff \exists t \in \mathit{Traces}(K)\colon t \dot{\models}^k \varphi$$

Proof. Analogous to Lemma 4.1. □

This version of the encoding will be used later as well. Based on the pessimistic encoding, we can now formulate the core theorem for LTL bounded model checking.

Theorem 4.3. *Let K be a system and let φ be an LTL formula.*

$$\left(\exists k.\, \mathrm{SAT}\left(\llbracket \neg\varphi \rrbracket^k\right)\right) \implies K \not\models \varphi$$

Proof. With Lemma 4.1, we obtain a trace t such that $t \models^k \neg\varphi$ for some bound k. Proposition 3.1 implies that $t \models \neg\varphi$ holds in the unbounded semantics which is equivalent to $t \not\models \varphi$ by the LTL semantics. Now that we have found a trace for which $t \models \varphi$ does not hold, we may conclude that $K \models \varphi$ does not hold as well. □

Since the LTL semantics is defined for infinite traces, there are some crucial limitations of BMC caused by the fact that we only consider trace prefixes. In the current encoding, we simply assume that the formula is not fulfilled at all if we did not find a witness within k steps. This is true in the optimistic semantics, but not in the general case.

Indeed, it is easy to come up with a formula φ whose negation holds without having a finite witness. Consider for example $\varphi \coloneqq \Diamond a$. In this case, a counterexample is an infinite path on which a never holds. The crucial point is that a counterexample may be infinite. The formula is not refutable in finite time. Consequentially, we do not even obtain a full semi-decision procedure for disproving the validity of LTL formulas with this approach. The reverse direction of Theorem 4.3 does not hold. Before we solve this issue, we introduce BMC for HyperLTL which is based on the limited encoding we have seen until now.

4.2 HyperLTL Bounded Model Checking

Since model checking hyperproperties is very expensive in the general case, there is a quest for model checking algorithms that are at least efficient for some practically relevant problems. For LTL, bounded model checking is an algorithm that turned out to be very useful in practice, even though it is inherently incomplete. Since HyperLTL is just an extension of LTL, a natural question that emerges is whether we can lift BMC to HyperLTL as well.

This is indeed possible and was initially proposed by Hsu et al [10]. The overall idea is to reduce model checking to QBF solving (instead of SAT solving for LTL) such that we have a satisfiability solver that takes care of the quantifiers. Since the quantifiers are the only difference between LTL and HyperLTL, the basic translation is a straightforward extension of the LTL BMC encoding presented in the last section.

Let Φ be a HyperLTL formula such that $\Phi = Q_1\pi_1.\dots Q_n\pi_n.\varphi$, where φ is quantifier free. We define

$$[\![\Phi]\!]^k \coloneqq Q_1\alpha_1.\dots Q_n\alpha_n.[\![\varphi]\!]^k$$

where $\alpha_i \in Act^k$ for $i \in 1..=n$. Analogous to the LTL BMC case in the previous section, the action variables determine the paths of the quantified systems. Additionally, $[\![\dot{\Phi}]\!]^k$ is defined analogously based on the optimistic LTL encoding. Essentially all results of the previous section also apply to HyperLTL.

Lemma 4.4. *Let K be a system, let Φ be a HyperLTL formula and let $k \in \mathbb{N}$.*

$$\text{SAT}\left([\![\Phi]\!]^k\right) \iff K \models^k \Phi$$
$$\text{SAT}\left([\![\dot{\Phi}]\!]^k\right) \iff K \dot{\models}^k \Phi$$

Proof. Lemma 3 from [10]. We start with the inner LTL formula and observe that the lemmas for LTL BMC from the previous section hold analogously, except that we consider a tuple of traces instead of a single one. Then we argue inductively about adding quantifiers to the formula and eliminating them from the aforementioned tuple stepwise. □

Theorem 4.5. *Let K be a system and let Φ be a HyperLTL formula.*

$$\left(\exists k.\, \text{SAT}\left([\![\neg\Phi]\!]^k\right)\right) \implies K \not\models \Phi$$

Proof. Theorem 1 from [10] with the pessimistic semantics. If the negated formula is satisfied even in the bounded pessimistic semantics, then the same holds in the unbounded semantics. □

It is not surprising to observe that the reverse direction of the theorem does not hold since we used the same formula encoding as in Section 4.1.2. Using this result, it is only possible to find finite counterexamples.

Chapter 5

Completeness for LTL Bounded Model Checking

A complete bounded model checking procedure for a system K and a formula φ is a terminating program that outputs true if and only if $K \models \varphi$, while considering only finite executions. We address two different approaches leading to a complete BMC algorithm for LTL. The former introduces a completeness threshold, while the latter interprets finite paths as looping infinite paths in order to reason about the unbounded semantics.

5.1 Completeness Threshold

The conceptually easiest approach to obtain more results from the bounded encoding is to choose the unfolding bound sufficiently high such that we are able to deduce more properties. According to [30], a *completeness threshold* for LTL BMC is a natural number c such that the absence of a counterexample up to bound c proves that $K \models \varphi$. The intuition is that if we did not find a counterexample of length c, we have unfolded the system and the formula far enough such that we can be sure that a longer counterexample does not exist either. Such a completeness threshold must exist since we are dealing with finite state systems.

Finding the smallest completeness threshold is at least as hard as model checking. It is equal to the length of a shortest counterexample, or zero if none exists [30]. Thus, we aim for an over-approximation that is as small as possible. But the issue with (approximations of) completeness thresholds is that they are often huge in practice. Since LTL model checking is PSPACE-complete [31] and LTL BMC is in NP [9], a general completeness threshold is most likely exponential with respect to the size of the system and the formula. Thus, a completeness threshold is only relevant in practice if we impose some restrictions ensuring that the bound has a reasonable size. A fragment of LTL for which model checking is NP-complete consists of formulas involving only the temporal operators $\Box$ and $\Diamond$ [31], so this is a more promising starting point.

Formulas of the form $\Box\psi$ where ψ is a propositional formula are known to have a comparatively small completeness threshold, namely the reachability diameter [22].

Definition 5.1 (Reachability Diameter [22])**.** The *reachability diameter* $rd(K)$ is the minimal number of steps required to reach all states of the system.

$$rd(K) \coloneqq \min\left\{d \mid \forall s \in Reach(K).\exists d' \leq d.\exists \alpha \in Act^{d'}.s = \delta^{d'}(\alpha)\right\} \qquad \lrcorner$$

If we know that every reachable state satisfies ψ, then we can obviously conclude that every valid trace must satisfy $\Box\psi$. The reachability diameter can be computed by QBF solving based on the following alternative characterization.

$$rd(K) = \min\left\{d \mid \forall \alpha \in Act^{d+1}.\exists \alpha' \in Act^{d}.\bigvee_{i=0}^{d} \sigma[i] \Leftrightarrow \sigma'[d+1]\right\}$$

The intuition is to check incrementally whether every state reachable within $d+1$ steps can also be reached sooner. The largest d for which this does not hold any more is the reachability diameter.

For formulas of the form $\Diamond\psi$, the reachability diameter is not sufficient in terms of a completeness threshold. This is illustrated by Figure 5.1. In both transition systems, the reachability diameter is 1. Unfolded for only one step, both transition systems are equivalent. But only the right one satisfies the property $\Diamond a$.

Instead, it suffices to visit all loop-free paths, i.e. those paths that do not visit any state twice. The corresponding completeness threshold is called reachability recurrence diameter [32].

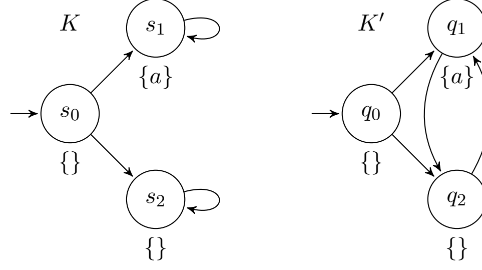

Figure 5.1: Two systems satisfying having the same reachability diameter $rd(K) = rd(K') = 1$, but a different reachability recurrence diameter $rrd(K) = 1 < 2 = rrd(K')$.

Definition 5.2 (Reachability Recurrence Diameter [32])**.** The *reachability recurrence diameter* $rrd(K)$ is the length of a longest loop-free path in the system.

$$rrd(K) \coloneqq \max \left\{ d \mid \exists \alpha \in Act^d. \bigwedge_{0 \leq i < j \leq d} \sigma[i] \not\Leftrightarrow \sigma[j] \right\}$$ ⌟

Note the difference from the reachability diameter at the example of Figure 5.1. The path $q_0 q_1 q_2$ is a longest loop-free path of K' while there is no loop-free path in K of the same size. It is worth mentioning that the reachability recurrence diameter is an over-approximation of the recurrence diameter. It is potentially much easier to compute since it does not involve any quantifier alternation.

Together with the encoding of Lemma 4.4, we obtain a complete BMC procedure for the $\Box\psi$ and $\Diamond\psi$ fragments of LTL. Note that we use a different bounded semantics in each case below since a globally-property is never fulfilled in the pessimistic semantics $\models^d$ and a finally-property is always fulfilled in the optimistic semantics $\dot{\models}^d$.

Theorem 5.1. *Let K be a system and let ψ be a propositional formula.*

1. *Let $\varphi \coloneqq \Box\psi$ and let $d \coloneqq rd(K)$.*

$$K \dot{\models}^d \varphi \iff K \models \varphi$$

2. *Let $\varphi \coloneqq \Diamond\psi$ and let $d \coloneqq rrd(K)$.*

$$K \models^d \varphi \iff K \models \varphi$$

Proof. One direction is given by Lemma 4.4 respectively.

1. $\implies$: By assumption, ψ holds on all reachable states. Thus it holds on every valid trace.

2. $\Longleftarrow$: Assume the contrary, that there is a trace t with $t \not\models^d \varphi$. By assumption, t contains a loop, i.e. there exist indices i, j with $t[i] = t[j]$. Now $t[..i](t[i..j])^\omega \not\models \varphi$, contradiction. $\square$

The work of Kroening et al. [33] discusses a larger fragment of LTL that has a completeness threshold which is linear in the recurrence diameter. They require that the LTL formula can be represented as a Büchi automaton that fulfils some additional requirements, namely that it can be decomposed into clique-shaped strongly connected components [33]. A theoretical completeness threshold for full LTL is presented in [30]. Note that a universal completeness threshold presupposes neither the optimistic nor the pessimistic encoding, since both are unable to detect infinite counterexamples in general. Instead it relies on the encoding presented in the following section, which incorporates a loop condition.

5.2 Loop Constraints

Now we consider a second approach that enables completeness. An open question that arose before was how to find infinite counterexamples via bounded model checking. For example, consider again system K from Figure 5.1. $K \models \Diamond a$ does not hold. A counterexample is a trace t satisfying the negated property $t \models \Box \neg a$. In general, it is impossible to determine from a trace prefix whether a globally property holds. Instead, we require a finite representation of infinite traces such that we can search for counterexamples by SAT solving in the manner of Theorem 4.3.

This motivates the idea of finding counterexamples consisting of a finite prefix, followed by a looping finite suffix. Note that we are talking about a path that must loop, not only the corresponding trace. A loop is characterized by a state that occurs twice in the path prefix. For instance, a counterexample for $K \models \Diamond a$ is an infinite path of the form $s_0(s_2)^\omega$. In this example, the path prefix $s_0 s_2 s_2$ is a finite representation of the infinite counterexample $s_0(s_2)^\omega$.

We may assume that the loop end is always determined by the last state of our path prefix. How to obtain the loop following the previous discussion is illustrated in Figure 5.2. If we go back to our formula encoding in Section 4.1.2, we can now evaluate the formula beyond the bound, assuming that we are given a looping path. In the notation of Figure 5.2, we know that $[\![\varphi]\!]_{k+1} = [\![\varphi]\!]_j$ for any sub-formula φ since $\sigma[(k+1)..] = \sigma[j..]$.

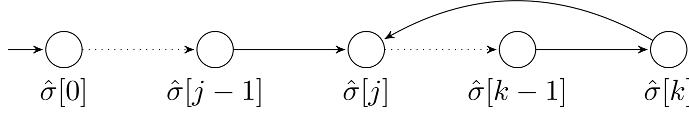

Figure 5.2: Loop interpretation of a finite path $\hat{\sigma} \in S^{k+1}$ with $\hat{\sigma}[j-1] = \hat{\sigma}[k]$ representing an infinite path $\sigma := \hat{\sigma}[..j](\hat{\sigma}[j..=k])^{\omega}$.

The complete encoding that we outline here is given in [19]. In contrast to the pessimistic encoding given in Section 3.5.1, only the rightmost column changes. Until and release can now be properly evaluated based on a fixed point characterization. It suffices to iterate once through the loop in order to evaluate these operations on the full infinite looping trace. This one iteration is captured by the sharp-bracket encoding. The sharp-bracket encoding ensures that we obtain a finite encoding by evaluating to fixed point values when exceeding the bound. We omit some easy cases of the encoding.

$:=$	$k' \leq k$	$k' > k$
$[\![a]\!]^k_{k'}$	$\mathcal{L}_a(\sigma[k'])$	$\bigvee_{j=1}^{k} \left(l^j \wedge \mathcal{L}_a(\sigma[j])\right)$
$[\![\bigcirc \varphi]\!]^k_{k'}$	$[\![\varphi]\!]^k_{k'+1}$	$\bigvee_{j=1}^{k} \left(l^j \wedge [\![\varphi]\!]^k_{j+1}\right)$
$[\![\varphi_1 \,\mathcal{U}\, \varphi_2]\!]^k_{k'}$	$[\![\varphi_2]\!]^k_{k'} \vee ([\![\varphi_1]\!]^k_{k'} \wedge [\![\varphi_1 \,\mathcal{U}\, \varphi_2]\!]^k_{k'+1})$	$\bigvee_{j=1}^{k} \left(l^j \wedge \langle\!\langle \varphi_1 \,\mathcal{U}\, \varphi_2 \rangle\!\rangle^k_j\right)$
$[\![\varphi_1 \,\mathcal{R}\, \varphi_2]\!]^k_{k'}$	$[\![\varphi_2]\!]^k_{k'} \wedge ([\![\varphi_1]\!]^k_{k'} \vee [\![\varphi_1 \,\mathcal{R}\, \varphi_2]\!]^k_{k'+1})$	$\bigvee_{j=1}^{k} \left(l^j \wedge \langle\!\langle \varphi_1 \,\mathcal{R}\, \varphi_2 \rangle\!\rangle^k_j\right)$
$\langle\!\langle \varphi_1 \,\mathcal{U}\, \varphi_2 \rangle\!\rangle^k_{k'}$	$[\![\varphi_2]\!]^k_{k'} \vee ([\![\varphi_1]\!]^k_{k'} \wedge \langle\!\langle \varphi_1 \,\mathcal{U}\, \varphi_2 \rangle\!\rangle^k_{k'+1})$	*false*
$\langle\!\langle \varphi_1 \,\mathcal{R}\, \varphi_2 \rangle\!\rangle^k_{k'}$	$[\![\varphi_2]\!]^k_{k'} \wedge ([\![\varphi_1]\!]^k_{k'} \vee \langle\!\langle \varphi_1 \,\mathcal{R}\, \varphi_2 \rangle\!\rangle^k_{k'+1})$	*true*

The variable l^j marks the state in the loop following the duplicate, $l^j = 1$ only if $\sigma[j-1] = \sigma[k]$ for $j \in 1..=k$. In Figure 5.2, precisely the jth position is marked. Note that if there is no loop, the encoding is equivalent to the old encoding from section 4.1.2. The loop variables l^j are fixed by a separate encoding LC^k (loop constraint) that is conjoined with the actual formula encoding.

$$\begin{aligned}
LC^k &:= Loop^k \wedge AtMostOne^k \\
Loop^k &:= \bigwedge_{j=1}^{k} (l^j \Rightarrow (\sigma[j-1] = \sigma[k])) \\
AtMostOne^k &:= \bigwedge_{j=1}^{k-1} (InLoop^j \Rightarrow \neg l^{j+1}) \\
InLoop^k &:= \bigvee_{j=1}^{k} l^j
\end{aligned}$$

Overall, we obtain the encoding $[\![\varphi]\!]^k := [\![\varphi]\!]_0^k \wedge LC^k$ for a HyperLTL formula φ. For the new encoding, both directions of Theorem 4.3 hold.

Lemma 5.2. *Let K be a system, let φ be an LTL formula.*

$$\left(\exists k.\, \mathrm{SAT}\left([\![\varphi]\!]^k\right)\right) \iff \exists t \in Traces(K)\colon t \models \varphi$$

Proof. Theorem 1 and the comment below in [19]. □

Theorem 5.3. *Let K be a system, let φ be an LTL formula.*

$$\left(\exists k.\, \mathrm{SAT}\left([\![\neg\varphi]\!]^k\right)\right) \iff K \not\models \varphi$$

Proof. Immediate consequence of Lemma 5.2. □

The above does not yet suffice to achieve completeness, but at least Theorem 5.3 suffices to disprove all invalid formulas. Proving $K \models \varphi$ based on this Theorem would require to verify that $[\![\neg\varphi]\!]^k$ is unsatisfied for all k. Of course, we cannot evaluate infinitely many SAT problems in practice. How to obtain a complete procedure based on the preceding is for example described in [34]. The fundamental idea is to unfold until all loop-free path prefixes that potentially violate the formula have been considered. But instead of estimating a huge completeness threshold, the proof is established by an incremental algorithm. We abstain from discussing the procedure in full detail here since it cannot be generalized to HyperLTL for reasons discussed later in Section 6.3.

Nevertheless, the conceptual idea of utilising loops to represent infinite paths by finite path prefixes is essential when it comes to arguing about infinite witnesses. We will encounter a generalized version of the loop constraint in Section 6.5.

Chapter 6

Completeness for HyperLTL Bounded Model Checking

Now we go one step further and aim for a complete decision procedure for HyperLTL, based on what we learned about LTL in the previous chapter. In the following, we pursue several different approaches that are all applicable to a fragment of HyperLTL respectively. Those fragments are identified by the shape of the quantifier prefix and the kind of the inner LTL formula. In this thesis, we focus on HyperLTL formulas with one quantifier alternation, followed by an LTL invariant.

At first, we consider alternation-free HyperLTL since this is arguably the easiest case as mentioned in Chapter 2. After this, we discuss why a completeness threshold must exist in general and provide a theoretical completeness threshold derived from automaton-based model checking in Section 6.2. Since this quickly leads to an infeasible procedure entailing a huge bound, we continue looking for more tight bounds. We examine whether techniques from LTL BMC are applicable here. To this end, we highlight the differences between LTL and HyperLTL that are relevant in the context of bounded model checking in Section 6.3. It turns out that the LTL procedure cannot be easily generalized to HyperLTL which is why we propose alternative approaches for fragments of HyperLTL in Sections 6.4 and 6.5.

6.1 Alternation-free Fragment

First, we consider the alternation-free fragment of HyperLTL, i.e. the fragment consisting of all formulas with a quantifier prefix of the form $\forall^n$ or $\exists^n$ for some $n \in \mathbb{N}$. As discussed in Chapter 2, model checking is much easier for formulas incorporating no quantifier alternation. We review it here primarily to illustrate the concept of composition to which we refer several times later.

Model checking alternation-free formulas can be accomplished by constructing a single composed system out of all quantified systems. Instead of considering n traces, we consider a single trace over n-tuples. This implies that we can replace the totality of all trace assignments by one single trace assignment in the composed system such that HyperLTL model checking is finally reduced to LTL model checking. An automaton-based model checking algorithm relying on this construction is presented in [11]. We adapt this procedure to our model checking setting.

Definition 6.1 (Composition of systems)**.** Let $K_i \coloneqq (S_i, s_0^i, Act_i, \delta_i, AP_i, L_i)$ be systems for $i \in ..n$. W.l.o.g. we assume that the sets of atomic propositions are pairwise disjunct. Based on this, we define a *composed system* $\times_{i\in..n} K_i \coloneqq (S, s_0, Act, \delta, AP, L)$, where

- $S \coloneqq \times_{i\in..n} S_i$,
- $s_0 \coloneqq (s_0^0, \ldots, s_0^{n-1})$,
- $Act \coloneqq \times_{i\in..n} Act_i$,
- $\delta(s) \coloneqq (\delta_0(s[0]), \ldots, \delta_{n-1}(s[n-1]))$,
- $AP \coloneqq \bigcup_{i\in..n} AP_i$,
- $L(s) \coloneqq \bigcup_{i\in..n} L_i(s[i])$. ⌟

Proposition 6.1. *Let $K = (K_1, \ldots, K_n)$ be a sequence systems and let $\Phi = Q\pi_1 \ldots Q\pi_n.\varphi$ be a HyperLTL formula with $Q \in \{\forall, \exists\}$.*

$$K \models \Phi \iff \times_{i\in..n} K_i \models Q\pi.\varphi'. \quad ⌟$$

Remember that model checking a HyperLTL formula of the form $\forall\pi.\varphi$, where φ is an LTL formula, is equivalent to LTL model checking of φ. Thus, we can apply LTL bounded model checking as discussed in Chapter 5 to verify a HyperLTL formula containing only universal quantifiers.

Due to Proposition 3.3, formulas containing only existential quantifiers can be handled analogously. Negating a formula in the $\exists^n$-fragment results in a formula in the $\forall^n$-fragment that is satisfied if and only if the original formula is not satisfied.

Note that a generalization this result enables us to reduce any sequence of consecutive quantifiers of the same sort in the quantifier prefix to a single quantifier of the respective sort. Hence, it suffices to differentiate between quantifier prefixes only based on their quantifier alternations in terms of model checking.

6.2 Automaton-Based Completeness Threshold

Now, we aim to come up with a completeness threshold for formulas with quantifier alternation. Remember that we considered a completeness threshold in Section 5.1 as a bound, such that evaluating the formula in some bounded semantics allows to conclude whether or not the formula holds in the unbounded semantics. For simplicity, we restrict ourselves to HyperLTL formulas incorporating an LTL invariant, as a generalization of what was discussed for LTL in Section 5.1. The concept described here can potentially be generalized to full HyperLTL, but this presupposes yet another bounded semantics. Neither the pessimistic nor the optimistic semantics behave like the unbounded semantics, even for an infinitely large bound. This is because the former assumes $\Diamond$-properties to be always true while the latter evaluates $\Box$-properties to false. Against this background, a bounded semantics which behaves optimistically on global requirements and pessimistically on instantaneous requirements would be a promising choice.

Our first approach for obtaining a completeness threshold relies on automaton-based model checking for HyperLTL. In fact, the completeness threshold for LTL determined in Section 5.1 may be derived analogously from automaton-based LTL model checking. This has been proposed by Clarke et al. and leads to a completeness threshold for full LTL [30]. We apply the same idea to HyperLTL.

The automaton-based model checking procedure we present here is essentially a special case of the model checking algorithm for HyperCTL* given in [11]. The construction assumes that the HyperLTL formula Φ contains only $\exists$-quantifiers that may be preceded by a negation. It is possible to transform an arbitrary HyperLTL formula into this form by applying the negation rule for quantifiers multiple times and dropping double negations between quantifiers afterwards. In our case, the inner LTL formula is always of the form $\Box\psi$ or $\Diamond\psi$ after applying this transformation, where ψ is a propositional formula. In order to prove whether or not Φ holds, we aim to construct an automaton whose language is not empty if and only if the property is satisfied. The algorithm building this automaton starts with an automaton for the inner LTL formula and eliminates one quantifier per step subsequently from the inside to the outside. For the remaining of this section, we assume again that $K \coloneqq (K_1, \ldots, K_n)$ is the associated sequence of systems, where $n \in \mathbb{N}$ is the number of quantifiers in Φ.

The initial automaton $\mathcal{A}_n$ is defined as a Büchi automaton accepting the same language as the inner LTL formula. This construction is standard [15]. The language of this automaton is $\mathcal{P}(\bigcup_{j\in 1..=n} AP_j)$. Note that we assume that the sets AP_j are pairwise disjunct. The quantifiers of Φ are eliminated consecutively by projection. Consider the step where $\exists\pi_i$ is eliminated, given $i \in 1..=n$. By induction, we have already constructed an automaton $\mathcal{A}_i$ over the alphabet $\mathcal{P}(\bigcup_{j\in 1..=i} AP_j)$. Let $\mathcal{A}_{K_i}$ be a Büchi automaton over the same alphabet such that $w \in \mathcal{L}(\mathcal{A}_{K_i})$ if and only if $w|_i$ is a valid trace of K_i. Now we define $\mathcal{A}'_{i-1} \coloneqq (\mathcal{A}_i \cap \mathcal{A}_{K_i})|_{1..i}$. If a negation precedes the existential quantifier, we define $\mathcal{A}_{i-1} \coloneqq \overline{\mathcal{A}'_{i-1}}$, otherwise $\mathcal{A}_{i-1} \coloneqq \mathcal{A}'_{i-1}$. Note that the projection onto system indices above is notation for the projection onto the respective sets of atomic propositions. In the end, we obtain an automaton $\mathcal{A}_0$ over an one-letter alphabet. We argue that Φ is satisfied if and only if the language of $\mathcal{A}_0$ is not empty.

Overall, each automaton $\mathcal{A}_i$ represents exactly the language of Φ with the first i quantifiers removed.

Lemma 6.2. *Let Φ_i be the formula obtained from Φ by dropping the first i components of the quantifier prefix.*

$$w \in \mathcal{L}(\mathcal{A}_i) \iff w \models \Phi_i$$

where $w \models \Phi_i$ is notation for $\emptyset[\pi_1 \mapsto w|_1] \dots [\pi_i \mapsto w|_i] \models \Phi_i$.

Proof. By induction. The base case $i = n$ holds by definition of $\mathcal{A}_n$. Let $i \leq n$. By the inductive hypothesis, $\mathcal{A}_i \cap \mathcal{A}_{K_i}$ accepts all words w over the alphabet $\mathcal{P}(\bigcup_{j\in 1..=i} AP_j)$ incorporating a valid trace $w|_i$ of K_i that satisfy $w \models \Phi_i$. The projection $(\mathcal{A}_i \cap \mathcal{A}_{K_i})|_{1..i}$ results in an automaton accepting words w over the smaller alphabet $\mathcal{P}(\bigcup_{j\in 1..i} AP_j)$. w is accepted if and only if a valid trace t for K_i exists such that $w[\pi_i \mapsto t] \models \Phi_{i+1}$, or equivalent $w \models \exists\pi_i.\Phi_i$. Furthermore, $w \models \neg\exists\pi_i.\Phi_i$ holds for each $w \in \mathcal{L}(\overline{((\mathcal{A}_i \cap \mathcal{A}_{K_i})|_{1..i})})$. Considering the definitions of Φ_{i-1} and $\mathcal{A}_{i-1}$, it is easy to see that combining both cases leads to $w \models \Phi_{i-1}$ for each $w \in \mathcal{L}(\mathcal{A}_{i-1})$. ☐

Theorem 6.3.

$$\mathcal{L}(\mathcal{A}_0) \neq \emptyset \iff K \models \Phi$$

Proof. Special case of Lemma 6.2 with $i = 0$. ☐

The automata obtained from the construction above fulfil some characteristic that simplifies arguing about them.

Lemma 6.4. *All automata $\mathcal{A}_i$ from above are either safety or co-safety automata.*

Proof. By induction. The base case is easy. $\mathcal{A}_n$ is obviously a safety automaton since the inner formula is an invariant. In the other case, its negation is a safety automaton such that $\mathcal{A}_n$ is a co-safety automaton.

The inductive case is easy if $\mathcal{A}_i$ is a safety automaton, since safety is closed under intersection and projection preserves safety. Assume that $\mathcal{A}_i$ is a co-safety automaton. We claim that $\mathcal{A}'_{i-1} = (\mathcal{A}_i \cap \mathcal{A}_{K_i})|_{1..i}$ is a co-safety automaton as well in this case. It suffices to show that each $w \in \mathcal{L}(\mathcal{A}'_{i-1})$ has a good prefix. We argue that $\hat{w}|_{1..i}$ is such a good prefix, where $\hat{w}$ is a good prefix of some $\bar{w} \in \mathcal{L}(A_i)$ with $\bar{w}|_{1..i} = w$. First assume that $\hat{w}|_i$ is not a valid trace prefix for K_i. Then all extensions $\bar{w}'$ (including $\bar{w}$) of $\hat{w}$ satisfy $\bar{w}' \notin \mathcal{L}(A_{K_i})$ and thus $\bar{w}|_{1..i} \notin \mathcal{L}(A'_{i-1})$. But this contradicts $w \in \mathcal{L}(A'_{i-1})$ such that we may always assume that $\hat{w}|_i$ is a valid trace prefix for K_i. Now let $\bar{w}'$ be an extension of $\hat{w}$ such that $\bar{w}'|_i$ is a valid trace for K_i. Note that $\bar{w}'|_{1..i}$ may be an arbitrary extension of $\hat{w}|_{1..i}$, so it remains to show that $\bar{w}'|_{1..i} \in \mathcal{L}(\mathcal{A}'_{i-1})$ in order to prove that $\hat{w}|_{1..i}$ is a good prefix. We show the equivalent statement $\bar{w}' \in \mathcal{L}(\mathcal{A}_i \cap \mathcal{A}_{K_i})$. $\bar{w}' \in \mathcal{L}(\mathcal{A}_{K_i})$ holds by construction since $\bar{w}'|_i$ is a valid trace for K_i. Moreover, $\bar{w}' \in \mathcal{L}(A_i)$ because $\bar{w}'$ extends the good prefix $\hat{w}$ of $\mathcal{A}_i$.

Given that $\mathcal{A}'_{i-1}$ is a safety or co-safety automaton, it is obvious that the same holds for $\mathcal{A}_{i-1}$. To handle the case where the ith quantifier is preceded by a negation, it suffices to see that safety and co-safety are dual under complementation. □

The complexity of the overall procedure is clearly dominated by the automaton complementation caused by the quantifier alternation. But at least we avoid the doubly-exponential blow-up [16] by restricting the inner LTL formula to invariants. Complementing each individual (co-)safety automaton is exponential in the size of the automaton [35]. Constructing the union or intersection of two automata results in an automaton whose size is linear in the product [36]. The final non-emptiness check is decidable in linear time as well [36].

It is easy to see that the above construction introduces essentially one automaton complementation per quantifier alternation. Thus, each quantifier alternation adds one more exponent to the size of $\mathcal{A}_0$. This illustrates why quantifier alternation tends to have the biggest influence of the complexity of HyperLTL model checking

Remember that our initial objective was to derive a completeness threshold c from the automaton construction above. We argue that the reachability recurrence diameter of $\mathcal{A}_0$ is a completeness bound for HyperLTL BMC. The fundamental insight is that the non-emptiness problem for $\mathcal{A}_0$ may be decided by considering inputs of at most this length.

Lemma 6.5.

$$K \not\models \Phi \implies \exists d. K \not\models^d \Phi$$

Proof. Φ is always finitely refutable [37] since its inner LTL formula is an invariant. This is particularly justified by the automaton construction. Since $\mathcal{A}_0$ is finite, the nested DFS algorithm [38] deciding the emptiness of $\mathcal{A}_0$ is bounded in its depth. So given that $K \not\models \Phi$, it is possible to provide evidence for the invalidity by considering path prefixes whose length is bounded by the aforementioned bound. □

The following theorem characterises a completeness threshold.

Theorem 6.6. *Let $d - 1$ be the reachability recurrence diameter of $\mathcal{A}_0$.*

$$K \models^d \Phi \iff K \models \Phi$$

Proof. $\implies$: It suffices to show that $K \models^{d'} \Phi$ holds for all $d' > d$. Together with Proposition 3.6, we may apply the contraposition of Lemma 6.5 to reach the goal under this assumption. Assume to the contrary that $K \not\models^{d'} \Phi$ for some $d' > d$. Given that the property is violated, the automaton construction shows that it suffices to unroll the system up to bound d in order to establish a refutation. Deciding the non-emptiness of $\mathcal{A}_0$ is possible within bound d since the nested DFS algorithm [38] accomplishing this task either recurses or terminates as soon as both DFS stacks constitute a looping path. This implies $K \not\models^d \Phi$ which contradicts the assumption.
$\impliedby$: Lemma 4.4. □

The result is analogous to the completeness threshold for full LTL, stated as Theorem 1 in [30]. Note that the second bound specified there does not apply in our case since we require a proof of emptiness for the automaton, not only non-emptiness.

Even though we have a completeness threshold now, it is fairly useless in practice. Estimating the completeness thresholds leads to a number that is far too large. What we aim for as a bound is roughly some two-digit number, such that evaluating formula in the optimistic semantics is feasible. Even for only one quantifier alternation and LTL invariants this is unattainable due to the exponential complexity of automaton complementation. We can potentially reduce the bound significantly by computing $\mathcal{A}_0$ explicitly, but model checking by performing the final non-emptiness check is not more expensive than computing the reachability recurrence diameter in this case.

The findings above suggest that bounded model checking of HyperLTL with quantifier alternation is quite expensive if we aim for a completeness result. Nevertheless, we aim to invent some algorithms that are significantly more efficient than this approach and hopefully usable in practice. To this end, we discuss to what extent completeness approaches known from LTL BMC, which avoid computing a completeness threshold explicitly, may be utilised here.

6.3 Distinction from LTL Bounded Model Checking

One difficulty of BMC for HyperLTL originates from the fact that the notion of a witness is generalized. In the LTL case, BMC is just about showing whether or not a single path satisfying the formula exists. But in HyperLTL, we do not necessarily argue about a single path and most importantly not only about the existence due to quantifier alternation.

Consider for example a $\forall\exists$ formula. It is not obvious how a witness may be represented in this case. In the manner of the LTL approach, we could negate the formula to an $\exists\forall$ formula and look for a counterexample. A proof of the negated formula may consist of an appropriate existential path. But arguing why this path is actually a valid choice satisfying the LTL body is not trivial since the universal path variable is still unbound.

Just negating the HyperLTL formula does not necessarily simplify the problem. HyperLTL is closed under negation in the sense of Proposition 3.3, negating the formula only provides the dual model checking problem. Disproving an LTL formula by determining a counterexample is much easier than proving that it actually holds. The former is its primary scope of application. Such a distinction is not possible any more in HyperLTL. Instead, HyperLTL BMC is efficient if a bounded witness exists. This is the case if the LTL body does not impose any infinite requirements to the trace assignment, or equivalently if it is satisfiable under the pessimistic semantics. We discussed this special case in Section 4.2.

For LTL BMC, completeness is achieved by arguing why a counterexample path cannot exist. This way of thinking seems to be less convenient for HyperLTL. Continuing our example above, what does it mean that no counterexample to a $\forall\exists$ formula exists? It means that we cannot find a witness for the negated $\exists\forall$ formula. Due to the quantifier alternation, it seems to be impractical to apply a similar reasoning as in the LTL case. Considering all loop-free existential paths inspired by the LTL procedure is not necessarily sufficient since we are dealing with a further unbound path variable here.

Instead of negating the formula, we can aim to determine a witness for the original formula immediately. In fact, it suffices to develop an algorithm that provides a witness for any HyperLTL formula in order to achieve completeness.

Against this background, it is worth discussing how a witness for a $\forall\exists$ formula looks like. We cannot even fix the existential path in this case since it may be chosen depending on the universal path. A witness can be rather thought of as a function resolving the existential path. This idea can be generalized to multiple quantifier alternations.

Overall, determining witnesses and proving their validity is not easy for arbitrary HyperLTL formulas. Thus, we only consider syntactically restricted fragments of HyperLTL in the following. Those include only one quantifier alternation as well as an LTL invariant.

6.4 $\exists\forall$ + LTL Invariant

First, we consider HyperLTL formulas of the shape $\Phi = \exists \pi_1.\forall \pi_2.\square \psi$, where ψ is a propositional logic formula. For this and the subsequent section, we assume that $K := (K_1, K_2)$ is the associated sequence of systems.

6.4.1 Completeness Threshold

A completeness threshold in this case is a bound for which the optimistic semantics coincides with the standard semantics. We define it in terms of a solution to a QBF problem since this is how we aim to compute it. In particular, there is no need of computing an automaton explicitly any more. On the other hand, the method to obtain a completeness threshold described here exploits specifically the restricted formula shape and therefore does not generalize as well as the automaton-based approach.

Since our formula involves a globally operator, we require an infinite witness here. Thus we employ an idea known from LTL BMC, namely it is possible to interpret a looping path prefix as an infinite path. Since the formula contains an universal operator as well, there is an additional requirement to the looping path. The idea is that we must consider the looping existential path until we have observed all possible states the system could be in together with an universal path. Note that we are referring to the state of the composed system here, the state depends on both paths.

Formally, we claim that the minimal k for which

$$\text{UNSAT}\left(\exists \alpha_1.SP^k\right) \tag{6.1}$$

holds, where

$$SP^k := \bigwedge_{0 \leq i < j \leq k} (\sigma_1[i] \not\Leftrightarrow \sigma_1[j]) \vee (\varsigma_2[i] \not\Leftrightarrow \varsigma_2[j]), \tag{6.2}$$

is a completeness threshold. Remember that σ_1 is the path obtained by the execution α_1 for π_1 and ς_2 is the union of all paths σ_2 for π_2 as defined in Section 3.2. Intuitively, this property means that for any path σ_1, the path $zip(\sigma_1, \varsigma_2)$ loops. The simple path constraint SP^k is satisfied exactly by all loop-free paths of length k. Note the similarity to the reachability recurrence diameter. They are mostly identical, we are just considering some sort of a composed system here. Thus, this approach can be thought of as a generalization of the completeness threshold for LTL invariants discussed in Section 5.1. It is necessary to verify that the propositional formula holds on all reachable states.

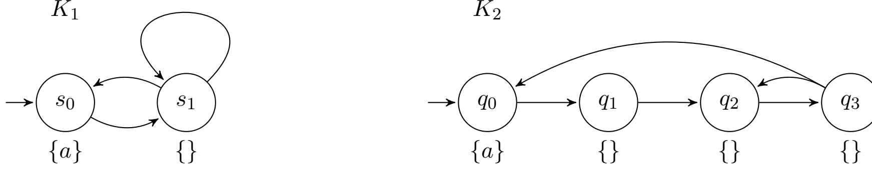

Figure 6.1: Systems satisfying the formula $\Phi \coloneqq \exists\pi_1.\forall\pi_2.\Box(a_{\pi_2} \to a_{\pi_1})$.

Example 6.1. We illustrate this approach at an example. Consider the systems given in Figure 6.1. We obtain $\varsigma_2 = \{q_0\}\{q_1\}\{q_2\}\{q_3\}(\{q_0, q_2\}\{q_1, q_3\})^\omega$. A maximal path prefix $\hat{\sigma}_1$ that does not loop simultaneously with ς_2 is $\hat{\sigma}_1 \coloneqq s_0 s_1 s_1 s_1 s_1 s_1 s_0$. Thus, $k \coloneqq 7$ is the minimal bound satisfying equation 6.1.

Consider the LTL formula $\Phi \coloneqq \exists\pi_1.\forall\pi_2.\Box(a_{\pi_2} \to a_{\pi_1})$. By QBF solving and Lemma 4.2 we obtain a path prefix $\hat{\sigma}_1$ that is a witness in the optimistic semantics for bound k, e.g. $\hat{\sigma}_1 \coloneqq s_0 s_1 s_1 s_1 s_0 s_1 s_0 s_1$. Equation 6.1 tells us that $\hat{\sigma}_1$ and ς_2 must loop together. Indeed we can observe that there exists a loop, e.g. the one obtained from the two repeating states marked below.

	0	1	2	3	4	5	6	7	...
$\hat{\sigma}_1$	s_0	s_1	s_1	s_1	s_0	s_1	s_0	s_1	...
ς_2	$\{q_0\}$	$\{q_1\}$	$\{q_2\}$	$\{q_3\}$	$\{q_0, q_2\}$	$\{q_1, q_3\}$	$\{q_0, q_2\}$	$\{q_1, q_3\}$	...

If we continue the above loop of $\hat{\sigma}_1$ to an infinite path, we do not reach any new states for any choice of σ_2 since ς_2 loops simultaneously. Now we found a valid choice for π_1 such that all states reachable together with any path assignment of π_2 satisfy the propositional formula. ⌟

In general, we can prove the following lemma.

Lemma 6.7. *Let a k satisfying equation 6.1 be given.*

$$K \models^k \Phi \implies K \models \Phi$$

Proof. Let a k that satisfies the assumption be given. Let $\hat{\sigma}_1$ be the path prefix of π_1 obtained as a witness of $K \dot{\models}^k \Phi$.

We have to show that there exists a path σ_1 such that $\emptyset[\pi_1 \mapsto Tr(\sigma_1)] \models \forall\pi_2.\Box\psi$. By assumption, there are indices $i, j \in ..{=}k$ where $i < j$ satisfying $\hat{\sigma}_1[i] = \hat{\sigma}_1[j]$ and $\varsigma_2[i] = \varsigma_2[j]$. We show that $\sigma_1 \coloneqq \hat{\sigma}_1[..i](\hat{\sigma}_1[i..j])^\omega$ fulfils the requirement.

First, we observe that ς_2 can be represented in a similar manner. The equality $\varsigma_2 = \varsigma_2[..i](\varsigma_2[i..j])^\omega$ follows inductively from the definition of ς_2. Each value in the sequence only depends on its immediate predecessor and we know that $\varsigma_2[i] = \varsigma_2[j]$.

Note that for each $\iota \in ..{=}j$ and $s \in \varsigma_2[\iota]$, the propositional formula ψ holds in state $(\sigma_1[\iota], s)$. By definition of ς_2, $s = \sigma_2[\iota]$ for some path σ_2. We know that the formula is satisfied in the bounded semantics up to bound $k \geq \iota$.

Now let σ_2 be an arbitrary path for π_2 and let $\iota \in \mathbb{N}_0$. By the HyperLTL semantics, we need to show that ψ holds in each state $(\sigma_1[\iota], \sigma_2[\iota])$. It suffices to show that such a state can be reached within at most k steps by choosing σ_2 appropriately. In this case, we already know that ψ must hold. This is an immediate consequence of the definition of σ_1 and the alternative representation of ς_2 mentioned before. Both are looping simultaneously with loop end $j \leq k$ such that there exists an $\iota' \in ..{=}j$ with $\sigma_1[\iota'] = \sigma_1[\iota]$ and $\varsigma_2[\iota'] = \varsigma_2[\iota]$. Note that $\sigma_2[\iota] \in \varsigma_2[\iota']$ is reachable within ι' steps by definition of ς_2. □

This leads to a complete model checking algorithm.

Theorem 6.8. *Let a k satisfying equation 6.1 be given.*

$$\mathrm{SAT}\left(\llbracket\dot{\Phi}\rrbracket^k\right) \iff K \models \Phi$$

Proof.

$\Longrightarrow$: Lemma 4.4 and Lemma 6.7.

$\Longleftarrow$: Proposition 3.5 and Lemma 4.4. □

There is still an issue with this procedure, namely the computation of k via equation 6.1. So far, we asserted that this is an ordinary QBF problem. This presupposes particularly that ς_2 can be encoded into a boolean formula. Even though this is possible, it is not advisable since each element of ς_2 is a set of states. In order to encode a set, we have to reserve one bit for each potential element which means one bit per state of K in our case. Usually, the state space of a system is huge such that this becomes quickly infeasible in practice. So we should rather pursue another strategy in this regard.

Since ς_2 is a sequence that does not contain any variables, we can precompute it with a standard graph algorithm and enumerate the sets of states occurring in ς_2. As mentioned in the proof of Lemma 6.7, ς_2 always ends in a loop. For our purposes it actually suffices to determine solely this loop. To this end, we compute ς_2 along its definition incrementally until we observe the first repetition of elements.

Given the aforementioned loop, we may eliminate ς_2 and a bunch of conjuncts from SP^k as an optimization. Assuming that l and e are the indices marking the loop start respectively end, we can rewrite SP^k as

$$SP^k = \bigwedge_{\substack{l \leq i < j \leq k \\ ((j-i) \bmod (e-l))=0}} (\sigma_1[i] \not\Leftrightarrow \sigma_1[j]).$$

Note that the completeness threshold computed in equation 6.1 only depends on the system structure and not at the formula. Thus, it is a completeness threshold for any invariant. If the systems are fixed, it suffices to compute the bound only once.

At first glance, it appears as if computing the completeness threshold is fairly easy since equation 6.1 is just a SAT-problem. In contrast, the formula encoding results in a QBF-problem with quantifier alternation. But we must acknowledge that computing the loop in ς_2 is not for free. It involves exploring the full state space. We continue this discussion in Section 7.2.

In any case, we should aim to keep the unfolding bound as low as possible. A potential improvement involves incorporating the additional information provided by the formula when computing the bound. This is what we do next.

6.4.2 Incremental Algorithm

In this section, we will combine the encoding given in equation 6.1 with the formula encoding such that we can potentially reduce the unfolding bound. We do not compute a completeness threshold explicitly any more, but rather give an incremental algorithm that unfolds the formula and the system until a witness has been found.

Note that equation 6.1 can be rephrased as SAT $\left(\forall \alpha_1 . \neg SP^k\right)$. Thus, the equation requires that all existential paths loop with all universal paths. If we content ourselves with a proof for $K \models \Phi$, we may weaken this requirement since we only need one appropriate existential path as a witness. Note that disproving $K \models \Phi$ is easy since the negated formula is covered by the approach described in Section 4.2.

An additional requirement for the existential path to be a witness is that it fulfils the propositional formula ψ together with any universal path up to bound k. This can be formalized as a merging of equation 6.1 and the optimistic formula encoding.

$$\mathrm{SAT}\left(\exists \alpha_1 . \left(\neg SP^k \wedge \forall \alpha_2 . [\![\dot{\Box} \psi]\!]\right)\right) \tag{6.3}$$

The definition of SP^k remains untouched. Again, we search for a minimal k satisfying this equation. Such a k is a proof for $K \models \Phi$.

Lemma 6.9. *There exists a k satisfying equation 6.3 if and only if $K \models \Phi$.*

Proof.

$\Longrightarrow$: Let a k that satisfies the assumption be given. Just as in Lemma 6.7, we obtain a looping path prefix $\hat{\sigma}_1$ from the encoding. Unrolling the loop leads to a path σ_1. By assumption, σ_1 together with any path for π_2 satisfy $\Box\psi$ up to bound k. Analogously to Lemma 6.7, any state reachable beyond bound k is also reachable within at most k steps such that $K \models \Phi$ follows.

$\Longleftarrow$: Since we only consider finite systems, the length of a non-looping path is bounded. So if we choose large enough, the statement reduces to the backwards direction of Theorem 6.8. □

Algorithm 1 states the complete model checking algorithm. Correctness and termination follow by Lemma 6.9, Lemma 4.4 and Proposition 3.4.

Algorithm 1 $\exists\forall$ + LTL invariant

for all $k \in 1..$ **do**
  if SAT $\left(\llbracket\neg\Phi\rrbracket^k\right)$ **then**
    return $K \not\models \Phi$
  else if SAT $\left(\exists\alpha_1.\left(\neg SP^k \wedge \forall\alpha_2.\llbracket\dot{\Box\psi}\rrbracket\right)\right)$ **then**
    return $K \models \Phi$
  end if
end for

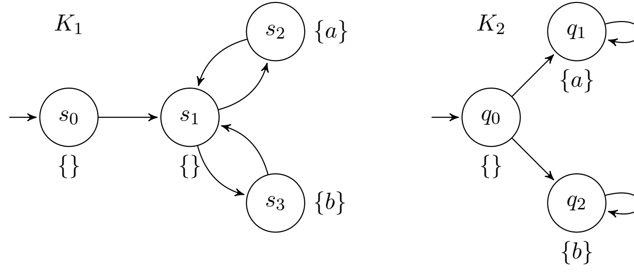

Figure 6.2: The formula $\forall\pi_1.\exists\pi_2.\Box(\bigwedge_{x\in\{a,b\}}\neg(x_{\pi_1}\wedge x_{\pi_2}))$ is violated. The shortest counterexample has length 4 which exceeds the reachability recurrence diameter by more than one.

6.5 $\forall\exists$ + LTL Invariant

For this section, we consider the case $\Phi = \forall\pi_1.\exists\pi_2.\Box\psi$, where ψ is a propositional logic formula.

Note that Theorem 6.8 does not hold in this case as witnessed by Figure 6.2. Consider the formula $\Phi = \forall\pi_1.\exists\pi_2.\Box(\bigwedge_{x\in\{a,b\}}\neg(x_{\pi_1}\wedge x_{\pi_2}))$. There exists no loop-free path of size 3, $k = 3$ satisfies equation 6.1. Nevertheless, there is no counterexample of length 3, but one of length 4. The path $\sigma_1 \coloneqq s_0 s_1 s_2 s_1 s_3$ cannot be mapped to a path σ_2 such that Φ is satisfied. So we need another approach.

Analogous to the previous section, it suffices to invent an algorithm that terminates if and only there Φ holds. Completeness is achieved by evaluating the negated formula in the optimistic semantics simultaneously.

Witness function

At first, we need to remind ourselves what constitutes a proof of satisfiability for a ∀∃ formula. Just as in the dual case, a witness should resolve the existential path quantifier. Notably, such a witness is not necessarily a single path in this setting. The choice of the existential path may depend on the choice of the universal path. Thus, a witness for $K \models \Phi$ is essentially a function $w\colon S_1^\omega \to S_2^\omega$ mapping each path σ_1 to a path σ_2 such that $\emptyset[\pi_1 \to Tr(\sigma_1), \pi_2 \to Tr(\sigma_2)] \models \Box\psi$.

Since the overall strategy is again to determine the witness via QBF solving, we require is a finite representation of such a witness function. We conjecture that w may be constructed from a finite function $\hat{w}\colon S_1^{k+1} \to S_2^{k+1}$, determined from an unrolling up to bound k. To this end, $\hat{w}$ must fulfil three characteristics which we describe next.

The first one is obvious, it ensures that the formula is actually fulfilled. So any two path prefixes $\hat{\sigma}_1$ and $\hat{w}(\hat{\sigma}_1)$ must fulfil the invariant ψ at each point in time up to bound k.

We want to be able to simulate w given $\hat{w}$ which means that for any σ_1, $w(\sigma_1)$ may be replaced by a repeated application of $\hat{w}$ on sub-paths of w. It remains to determine which sub-paths we actually mean here. This results from our iterative definition of w based on $\hat{w}$. Given σ_1, we first apply $\hat{w}$ to $\sigma_1[..{=}k]$. Now we assume that $zip(\sigma_1[..{=}k], \hat{w}(\sigma_1[..{=}k]))$ contains a loop, which we adopt as the second characteristic. This enables us to shorten σ_1 by cutting out precisely this loop. Now the procedure is repeated on the truncated σ_1.

The remaining challenge is to ensure that the path prefixes obtained from $\hat{w}$ as described before can actually be merged into an infinite path. Basically this requirement is exactly the third constraint. Intuitively, it states that after cutting out the loop of σ_1, applying $\hat{w}$ on the new prefix still provides the same path as in the previous iteration — of course modulo skipping that loop and being able to observe the next path segment instead.

Formula Encoding

Now it remains to translate all requirements into a single formula that is checked for satisfiability. This is actually the point where QBF does not meet our needs since we have to quantify over a function. Instead we require SMT solving here. But apart from exchanging the underlying satisfiability solver, this does not require any further alterations. Somewhat counter-intuitively, we translate the ∀∃ HyperLTL formula into an ∃∀ QBF formula. This is due to the fact that the existentially quantified variable is solely a function f, corresponding to the witness function $\hat{w}$. Note that f is a function over action variables in this context, but as usual those correspond to some path. The universal quantifier prefix comprises the action variables defining σ_1. The three constraints imposed on $\hat{w}$ are conjoined in the formula body.

Constraint number one just corresponds to the formula encoding as usual. Constraint number two is something we basically know already. It is the negation of the simple path constraint SP^k with respect to the composed system, ensuring that σ_1 and σ_2 loop together. The third constraint is somewhat more complicated. It argues about loops obtained from the second constraint which is why it is described as an implication of a loop constraint LC^k detecting the loop and the actual so-called extension constraint EC^k.

We introduce a set of boolean loop variables l_i, e_i marking the start respectively end of a loop in the composed system at step i. Now we say that for all loops, the extension constraint must be fulfilled. It states that any other path σ'_1 agreeing with σ_1 modulo the loop (captured by the $\mathit{SubPath}^k_{i,j}$ constraint) inserted into the witness function provides a path that continues the path obtained from σ_1.

Definition 6.2 (Encoding of the formula)**.**

$$[\![\Phi]\!]^k := \exists f \forall \alpha_1 \forall \ell \forall \alpha'_1 . \neg SP^k \wedge \bigwedge_{0 \leq i \leq k} [\![\psi]\!]^k_i \wedge \left(LC^k \Rightarrow EC^k\right)$$

$$SP^k := \bigwedge_{0 \leq i < j \leq k} \sigma[i] \neq \sigma[j]$$

$$\begin{aligned}
LC^k &:= \mathit{Loop}^k \wedge \mathit{AtMostOne}^k \wedge \mathit{InLoop}^{k-1} \\
\mathit{Loop}^k &:= \bigwedge_{0 \leq i < j \leq k} (l_i \wedge e_j \Rightarrow \sigma[i] = \sigma[j]) \\
\mathit{AtMostOne}^k &:= \bigwedge_{i=1}^{k-1} \left(\left(\mathit{InLoop}^{i-1} \Rightarrow \neg l_i\right) \wedge \left(\mathit{LoopEnd}^i \Rightarrow \neg e_{i+1}\right)\right) \\
\mathit{InLoop}^i &:= \bigvee_{j=0}^{i} l_j \\
\mathit{LoopEnd}^i &:= \bigvee_{j=1}^{i} e_j
\end{aligned}$$

$$\begin{aligned}
EC^k &:= \bigwedge_{0 \leq i < j \leq k} \left(l_i \wedge e_j \Rightarrow \left(\mathit{SubPath}^k_{i,j} \Rightarrow \sigma[k] = \sigma'[k-(j-i)]\right)\right) \\
\mathit{SubPath}^k_{i,j} &:= \bigwedge_{\kappa=0}^{k-(j-i)-1} \text{if}\, \kappa \geq i \,\text{then}\, \alpha'_1[\kappa] = \alpha_1[\kappa + (j-i)] \,\text{else}\, \alpha'_1[\kappa] = \alpha_1[\kappa]
\end{aligned}$$

In the above, k is the unrolling bound. f is the witness function defined on actions. α_1 and α_1' both are variables for actions taken by the universal path. The corresponding actions taken by the existential path are obtained by applying f on each of them respectively. ℓ is a place holder for all loop related variables. The path σ (in the composed system) is obtained from α_1 and $f(\alpha_1)$. It is in particular considered at the evaluation of ψ. The path σ' is obtained from α_1' and $f(\alpha_1')$. ⌟

Lemma 6.10. *The satisfiability of* Φ *in the above encoding constitutes a witness for the validity of the formula, i.e.*

$$\exists k.\, \mathrm{SAT}([\![\Phi]\!]^k) \implies K \models \Phi.$$

Proof. Let a k that satisfies the assumption be given. Additionally, we obtain a function $\hat{w}\colon S_1^{k+1} \to S_2^{k+1}$ by assumption that can be constructed from f. We define a function $w\colon S_1^\omega \to S_2^\omega$ mapping each path σ_1 to a path σ_2 such that $\emptyset[\pi_1 \to Tr(\sigma_1), \pi_2 \to Tr(\sigma_2)] \models \Box\psi$.

Let σ_1 be a path for π_1. We define a family of index sets J_i of size $k+1$ for $i \in \mathbb{N}_0$. Those describe sub-paths of σ_1. Let $J_0 \coloneqq ..{=}k$. We define a family of path prefixes σ_2^i of π_2 for $i \in \mathbb{N}_0$. Let $\sigma_2^0 \coloneqq \hat{w}(\sigma_1[J_0])$. Let $i \in \mathbb{N}_0$. By assumption, $zip(\sigma_1[J_i], \hat{w}(\sigma_1[J_i]))$ contains a loop. Let j, j' be indices marking the loop start and end such that $zip(\sigma_1[J_i], \hat{w}(\sigma_1[J_i]))[j] = zip(\sigma_1[J_i], \hat{w}(\sigma_1[J_i]))[j']$. We define J_{i+1} as J_i without the loop and extend the index sequence to length $k+1$, formally $J_{i+1} \coloneqq J_i[..j]J_i[j'..](J_i[k]+1..J_i[k]+1+(j'-j))$. Note that $\sigma_1[J_i]$ is always a valid path for π_1 by definition. We define $\sigma_2^{i+1} \coloneqq \sigma_2^i \hat{w}(\sigma_1[J_{i+1}])[-(j'-j)..]$. In order to prove that this is a valid path for π_2, we need to show that $\sigma_2^i[-1] = \hat{w}(\sigma_1[J_{i+1}])[-(j'-j)-1]$. This is exactly what the extension condition ensures. By definition $\sigma_2^i[-1] = \hat{w}(\sigma_1[J_i])[-1]$ and since J_{i+1} is an extension of J_i without a loop the extension constraint gives us $\hat{w}(\sigma_1[J_i])[-1] = \hat{w}(\sigma_1[J_{i+1}])[-(j'-j)-1]$.

We can now define $w(\sigma_1) \coloneqq \lim_{i\to\infty} \sigma_2^i$. What remains to show is that the invariant ψ holds in each state $(\sigma_1, w(\sigma_1))[i]$ for $i \geq 0$. Note that we defined $w(\sigma_1)$ such that $(\sigma_1, w(\sigma_1))[i] = (\sigma_1[J_j], \hat{w}(\sigma_1[J_j]))[j']$ for some j, j'. Now ψ holds in $(\sigma_1[J_j], \hat{w}(\sigma_1[J_j]))[j']$ by assumption. □

Note that the other direction of Lemma 6.10 does not hold. Consider Figure 6.3. A finite witness function $\hat{w}$ does not exist. For any bound k, there are two options how to map s_0^k without violating the formula. Either we map it to $q_0 q_1^{k-1}$, then s_0^k can be continued to $s_0^k s_2$ where b holds in the last state, but $q_0 q_1^{k-1}$ cannot be continued by a state where b holds as well. Mapping s_0^k to $q_0 q_2^{k-1}$ yields a symmetric case. Still, it is easy to convince oneself that the property actually holds. The issue is that in the HyperLTL semantics, the choice of the entire infinite universal path is already known at the time we chose the existential path.

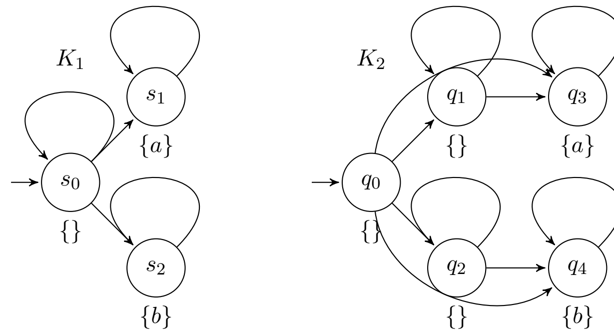

Figure 6.3: The formula $\Phi = \forall \pi_1 . \exists \pi_2 . \Box (\bigwedge_{x \in \{a,b\}} (x_{\pi_1} \leftrightarrow x_{\pi_2}))$ is satisfied, but no finite witness function exists.

Chapter 7

Evaluation

In the evaluation part, we focus on the results of Sections 6.4 and 6.5. For the alternation-free fragment, there already exists a tool that essentially implements the reduction to LTL model checking discussed in Section 6.1 [11, 17]. Estimating a completeness threshold based on the automaton construction from Section 6.2 quickly leads to a huge bound due to automaton complementation. This is unpracticable as discussed before, particularly since the number of quantified variables grows linearly with the bound.

All experiments were executed on a standard PC with a quad-core i5 and 16GB memory. Note that stated runtimes are just rough averages since the concrete values are fluctuating due to SMT solving. But for our purposes, it suffices to know the order of magnitudes anyway. Our primary mission is to determine to what extent model checking is feasible with our methods.

7.1 Implementation

We implemented our algorithms in Rust [39]. Systems are accepted in the Btor2 format [40]. Btor2 is a common word-level hardware model checking format, particularly utilised as a specification format for the hardware model checking competition (HWMCC) [41, 42] since 2019. Btor2 allows to represent bit-vector arithmetic and additionally features arrays. Its semantics follows closely the corresponding SMT-LIB logics and theories [24]. Thus, we make use of an SMT model checker instead of a QBF solver for simplicity.

```
1 sort bitvec 4
2 state 1 s
3 one 1
4 init 1 2 3
5 sort bitvec 1
6 input 5 i
7 uext 1 6 3
8 sll 1 2 7
9 next 1 2 8
```

```
s = b0001;
while (true) {
  i = getInput();
  s << i;
}
```

Figure 7.1: Bit-shift program. Left: BTOR2 specification. Right: Corresponding program in pseudocode.

7.1.1 Btor2

BTOR2 is a convenient specification language representing systems as defined in Section 3.2. Each line is identified by its line number and either evaluates to a sort or a value of a certain sort. The primary sort we consider here are bit-vectors, denoted as `bitvec <size>`. Aside from this, BTOR2 also supports arrays built from two sorts. All statements except of sorts are followed by a reference to the sort of their result, a potential list of arguments referring to other lines and finally an optional label. Each system state corresponds to the current value of all `state` bit-vectors combined and an action corresponds to all `input` bit-vectors combined. The initial state is defined by the `init` statements assigning each `state` an initial value. Similarly, the transition function is defined by the `next` statements assigning each `state` its successor value, based on the current `state` and `input` values.

A sample BTOR2 program is given in Figure 7.1.1. The state `s` is a 4-bit bit-vector that is initially one. In each step, we apply a left-shift by `i` positions, where `i` is a 1-bit input variable. For a more detailed description of the BTOR2 language refer to the official documentation [40].

7.1.2 Satisfiability Modulo Theories

In the previous chapters, we discussed primarily how to reduce model checking to QBF solving, including how to encode systems as QBF formulas. Since the BTOR2 systems operate on word-level, we chose to encode encode our problem in a little more high-level format supporting the respective language constructs natively. Another reason is that QBF solvers do not support functions as variables which we require for the algorithm discussed in Section 6.5.

Satisfiability modulo theories (SMT) is an extension of QBF allowing for more language constructs like bit-vectors and arithmetic. SMT-LIB is the de facto standard specification language for SMT problems. The BTOR2 language is by construction quite close to the SMT-LIB format with the bit-vector and array extensions. Unfolding an BTOR2 system analogous to Section 4.1.1 is straightforward.

State of the art satisfiability solvers for SMT problems are for instance Z3 [25] and cvc5 [26], which we utilized in our experiments. We primarily rely on Z3 and use the Z3 API in order to construct SMT expressions. All runtimes mentioned below have been obtained using Z3.

7.2 $\exists\forall$ + LTL Invariant

Initially, we analyse the model checking approaches presented in Section 6.4. So we are restricted to $\exists\forall$ formulas whose inner part is an LTL invariant. First of all, we require some reasonable benchmarks for this fragment. To the best of our knowledge, properties of this particular shape have not been considered in any related work. This suggests that our fragment may not be expressive enough to represent properties that are relevant in practice. In any case, we need to define our own problem statements.

The tasks we discuss in the following do not immediately intend to solve some real-world problems. The objective is rather to point out the capabilities and limitations of our algorithms. We will see in which cases applying our procedure is promising and in which cases it is nearly infeasible.

Overall, we focus on *planning* problems [43] in some form. For a general introduction into planning, please refer to [44]. In its original form, it is always about finding a sequence of actions (a plan) solving some task. It is related to our research in the sense that the desired plan may be obtained as the witness of some $\exists\forall$ formula, it corresponds to an appropriate choice for the existential path.

Utilising HyperLTL model checking in order to solve planning problems was first proposed in [3]. One particular example that is discussed in this work is optimal planning.

Example 7.1 (Optimal planning). *Optimal planning* is the task of finding a shortest plan in a system. In other words, an optimal plan is a plan such that all other plans do not reach the goal g until the optimal plan reaches the goal. This phrasing leads to the following HyperLTL formula, where both paths belong to the same system.

$$\exists\pi_1\forall\pi_2.\neg g_{\pi_2}\,\mathcal{U}\,g_{\pi_1}$$

Under the assumption that at least one plan exists, we can state the optimal planning problem in our fragment as well.

$$\exists \pi_1 . \forall \pi_2 . \Box (g_{\pi_2} \rightarrow g_{\pi_1})$$

We may verify whether there exists some plan by determining whether a witness to the preceding formula is actually a plan. Note that this formula is always satisfiable, we are just interested in the witness here.

Of course, both formalisations do not lead to an efficient planning algorithm at all. We could easily avoid the second path variable and instead increase the unrolling bound stepwise until the first plan has been found. The point is rather that HyperLTL is much more expressive and allows us to define more refined objectives, including further requirements to a path that may be attached to the formula given above. Another example are properties involving certain notions of robustness, but those do not belong to our fragment of HyperLTL [3]. ⌟

In the following, we first explain how to obtain planning problems that suit to our accepted specification language. Afterwards, we introduce several concrete planning settings and examine how our approach performs in each of them.

7.2.1 System Representation

The *planning domain definition language* (PDDL) is the de facto standard specification language for planning problems [45]. It separates a planning problem into a general domain file and a specific instance file. The domain defines a set of atomic propositions respectively predicates constituting the system state. The system state may proceed by taking an action. An action defines a requirement to the current state (precondition) enabling the action as well as a set of changes (effect) applied to the state as part of the transition to the next state. A concrete problem instance must only define an initial state and a goal state.

A simple PDDL specification modelling a vehicle is given in Figure 7.2. PDDL is a huge specification language with various features, but we limit ourselves to language constructs appearing in this example. This and some other planning problems we consider here are adapted from benchmarks used by the international planning competition [46].

Encoding

In order to apply our model checking algorithm to such a planning problem, we first need to encode and unroll the PDDL problem into Btor2. A general translation may be obtained as follows.

```
(define (domain drive)
  (:types place vehicle
         - object)
  (:predicates
    (road ?p1 ?p2 - place)
    (at ?v - vehicle
        ?p - place)
  )
  (:action drive
    :parameters
        (?v - vehicle
         ?p1 ?p2 - place)
    :precondition (and
      (at ?v ?p1)
      (road ?p1 ?p2)
    )
    :effect (and
      (not (at ?v ?p1))
      (at ?v ?p2)
    )
  )
)
```

```
(define (problem A-D)
  (:domain drive)
  (:objects
    A - place
    B - place
    C - place
    D - place
    truck - vehicle
  )
  (:init
    (road A B)
    (road B A)
    (road B C)
    (road C B)
    (road C D)
    (road D C)
    (at truck A)
  )
  (:goal (and
    (at truck D)
  ))
)
```

Figure 7.2: Left: PDDL domain. We have roads on which vehicles can drive from one place to another Right: PDDL problem. A truck must drive from place `A` to place `D`.

Objects of each type are represented as bit-vectors respectively. Additionally, we introduce one action type consisting of all actions, encoded as bit-vectors. An action variable as well as a set of variables representing the parameters for all actions constitute the BTOR2 inputs. A crucial question is how to store the state, capturing which predicates hold at the current point in time. Note that predicates may have multiple arguments. Thus, an obvious attempt is to keep them in multidimensional arrays where each argument represents one axis. BTOR2 allows to define such nested bit-vector array sorts. Checking whether an atomic proposition holds in some state corresponds to reading from an array.

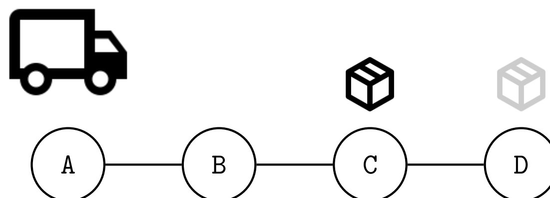

Figure 7.3: Transport planning problem. The goal is to deliver the package from place `C` to place `D` and return to place `A` with the truck.

It turns out that the support of SMT solvers like Z3 and cvc5 for multidimensional bit-vector arrays is limited. Some of our experiments led to inconsistent results. In any case, working with arrays is quite inefficient for our purposes. This is partly due to the fact that each write operation kind of creates a new array. According to a Z3 developer, it is advisable to omit fixed-size arrays completely and replace them by a sequence of single bits [47]. Usually, SMT solvers are much more optimized for the bit(-vector) logic. While an array representation may be more concise, it leads to harder SMT problems.

There is still one point concerning the encoding that deserves further consideration. What happens if the inputs in the encoded Btor2 problem are chosen such that they do not correspond to a valid action? It is conceivable that the precondition of the selected action is not fulfilled. Or the inputs representing the parameters possibly do not correspond to any valid object at all. In both cases, we determine that the encoded system performs implicitly a stutter step, i.e. it remains in the same state. In order to keep the behaviour of the encoded system consistent with the PDDL system, we assume that our domains include a stuttering action. The stuttering action has no precondition and no effect, thus it is always applicable and does not alter the state.

7.2.2 Full Example

Initially, we consider an extended version of the PDDL problem given in Figure 7.2 as a running example. In addition to the given specification, we introduce packages that may be transported by vehicles from one place to another. This requires a new package type as well as two new predicates, capturing whether a package is at some place or inside a truck. Additionally, we need two new actions, namely pickup and drop, defined along their natural meaning respectively. The goal is to deliver the package from place `C` to place `D` and to return the truck back to position `A`. A graphical representation of this scenario is given in Figure 7.3.

Below, we go through the different stages and variations of our approach stepwise.

Reachability Loop

The first thing we need to do in any case is determining the loop in ς_2 (in the notation of Section 6.4). In practice, this means that we need to simulate a system execution stepwise in order to compute ς_2. An exhaustive BTOR2 simulator already exists as part of the Btor2Tools [48]. This one is rather meant to be a witness verification tool for hardware model checking benchmarks, but we could adapt it to our purposes. Alternatively, it is possible to perform the simulation based on SMT solving on the encoded system.

Since we primarily consider planning benchmarks here, it is possible to avoid the overhead caused by the problem encoding at this stage. The reachability analysis may be performed immediately on the PDDL model which tends to be more efficient.

In our example, a repetition in the set of reachable states loops occurs after ten steps. The loop is described by the tuple $(start, end) \coloneqq (9, 10)$. It turns out that the loop diameter of one occurs in general for all our planning problems due to the introduced stutter action. This action implies that an atomic proposition reachable within exactly i steps is reachable within any greater number of steps as well by simply taking the stutter action. Thus, the loop start always corresponds to the reachability diameter. A most distant state in our example is the state where the package is at place `A` while the truck is at place `D`. A shortest path reaching this state is achieved by picking up the package at `C`, afterwards dropping it at `A` and finally driving the truck to `D`. We visit nine different states on this path as expected.

This observation suggests that it may not always be desirable to have some stutter action included into the specification. We continue this discussion later when considering an example without such an action.

Completeness Threshold

The full algorithm essentially consists of three steps. First, we need to find a loop in ς_2, which is exactly what we discussed just before. Next, we determine the completeness threshold as the minimal bound satisfying equation 6.1. Finally, we evaluate the HyperLTL formula in the optimistic semantics for this bound.

At first glance, it seems to be fairly easy to evaluate equation 6.1 since it only involves a prefix of existential quantifiers. On the one hand this is indeed true, but at the other hand we must acknowledge that the loop computation in the preceding step which eliminates the second quantifier is not for free. The expected runtime of the individual stages will be compared shortly.

We obtain 21 as a completeness threshold for our transport example. Again, we can interpret this number. Due to the loop diameter one, the right disjunct of equation 6.2 is always false for $i, j \geq \mathit{start}$. If we would omit this right disjunct entirely, we would obtain essentially the definition of the reachability recurrence diameter. Combining our observations leads to the insight that the completeness threshold is precisely the reachability diameter plus the maximum reachability recurrence diameter for any reachable state as initial state (plus one technically). This is indeed what we observe in our example. If truck and package are initially located at the most distant positions, we may visit eleven states without looping by transporting the package to the opposite location and returning the truck to the initial place.

Now that we have computed a completeness threshold, the final step of evaluating the HyperLTL formula in the optimistic semantics is straightforward. Remarkably, this is the only step in the whole procedure that requires solving an SMT problem involving quantifier alternation. Still this does not necessarily imply that it is the bottleneck of the whole procedure.

Incremental Algorithm

A completeness threshold of 21 for our small example in Figure 7.2 seems to be quite large. This is partly due to the fact that the completeness threshold is valid for the system specification independent of the HyperLTL formula as discussed before. This is why we invented a modified algorithm in Section 6.4.2 which takes into account the formula while computing the completeness bound.

Step one as described in the previous paragraph remains the same, but now we merge the last two steps into one. Thus, we check immediately an SMT formula with one quantifier alternation. In our running example, we find a proof at bound 10 which is a significant improvement. Note that this value coincides with the loop end of ς_2 which is not a coincidence. This is actually the least possible bound with respect to solely the simple path constraint. In the context of optimal planning it is quite obvious that unrolling beyond the reachability diameter is not necessary. The goal of planning is just reaching a goal state such that an optimal plan cannot be longer than the reachability diameter.

7.2.3 More Benchmarks

Now, we consider some more problem settings in order to examine the behaviour of our algorithms. The detailed PDDL files are enclosed, we only highlight the most important characteristics here.

	`4-1-1`	`8-1-1`	`4-2-1`	`4-1-2`	`8-2-2`
comp. t.	21 $(2s)$	38 $(143s)$	? $(\gg 1h)$	? $(\gg 1h)$	? $(\gg 1h)$
incr.	10 $(< 1s)$	15 $(2s)$	12 $(< 1s)$	13 $(< 1s)$	21 $(90s)$
optimal	8	10	6	10	13

Figure 7.4: Transport experiments. The last row shows the length of an optimal path and the two preceding rows indicate the unrolling bounds of the two algorithms with their runtime.

Transport

We start by considering some variations of the transport task from the previous section, still with the optimal planning objective. The complexity of the transport task mainly depends on the number of places, trucks and packages. Therefore, the benchmarks are named accordingly `places-trucks-packages`, where each name stands for the respective quantity. The results are given in Figure 7.4.

It turns out that increasing the number of packages respectively trucks increases the completeness threshold significantly such that even computing the completeness threshold itself becomes quickly infeasible. The size of the SMT problem to solve during the computation of the completeness threshold grows linearly, but still this is only feasible for small unfolding bounds. Even for seemingly small problems, the reachability recurrence diameter (and thereby the completeness threshold) can become quite large.

With our second approach, we can solve more problems since the completeness bound does not grow that fast. It is usually not much larger than the size of an optimal path. Still it takes some computational effort to detect the loop in ς_2. Exactly this is indeed the bottleneck if we try to increase the problem size further. Even for the `8-2-2` benchmark, there are already several thousand reachable states. We empathise this issue by considering another domain in a moment.

In the tasks involving multiple vehicles, we may verify whether a second vehicle is actually required to reach the optimal solution. This is just an example for a variation of a classical planning task that can be easily achieved by modifying the HyperLTL formula. It is accomplished by extending the invariant with another constraint, namely that one vehicle remains in its initial position. Note that this stricter requirement influences neither the bound nor the runtime for the examples discussed above.

Visit All

Next, we consider another popular planning domain. It features a robot moving in a grid-world. The robot is obliged to visit a number of states. Most notably, it must remember all states it already visited. This domain has the characteristic that even seemingly simple problem instances entail a huge state space. In our approach, this is a major issue since determining a loop in ς_2 becomes more difficult. In a 3×3 grid with roughly 800 reachable states, it is still possible to find a loop within a few seconds. But already in a 4×4 grid-world we have more than 50 000 reachable states. All of them are reachable within 23 steps. It takes us around 10 minutes to obtain the loop in this case. Computing the completeness threshold is obviously nearly impossible against this background. It is to be expected that the reachability recurrence diameter is large. However, obtaining a solution via the incremental algorithm is possible in this case since the bound is not too large. It takes less then a minute given that the loop is already known.

This illustrates a fundamental drawback of our algorithm. If there are lots of reachable states in the universally quantified system, the model checking procedure is infeasible. But whenever the universal path is somewhat more restricted, computing the loop is very likely to be feasible. The same potentially holds for the entire incremental algorithm under this assumption.

State Space Analysis

One observation we made during the previous tasks is that our procedure is definitively not efficient for optimal planning. However, the expressiveness of HyperLTL fragment goes far beyond optimal planning. Particularly in the planning domain, there are several other interesting properties that we can specify as HyperLTL formulas [3]. For instance, we are able to analyse the state space in more detail.

A concrete problem in this context is to examine whether there exists a plan that seems to be optimal at each intermediate position according to some heuristic function. Heuristic search is a common concept in planning [44]. Generally speaking, it entails a stepwise construction of a plan such that we approach the goal more closely in each step. The proximity to the goal is defined by a heuristic function mapping a state to a natural number. An optimal plan often has to include steps that seemingly go in the wrong direction with respect to a specific heuristic. The motivation is to check whether some greedy algorithm could be misled to take a non-optimal path.

To this end, we consider a modified version of the grid-world from above that includes walls which prevent certain transitions. Moreover, the goal of the robot is only to reach a single goal state. Note that we do not need to store the visited states any more such that the state space reduces significantly. An appropriate heuristic function h in this case simply calculates the Hamming

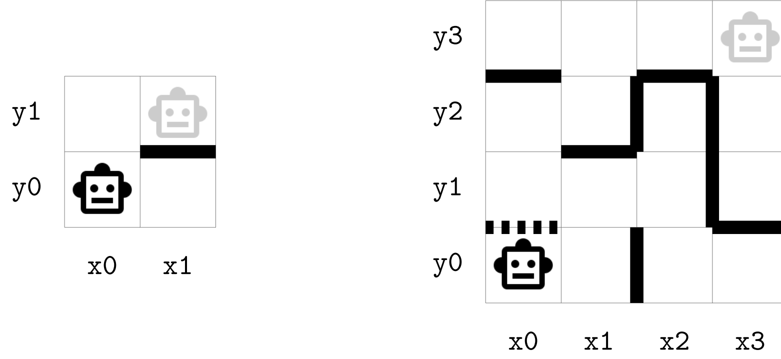

Figure 7.5: Grid-world problem instances. The transparent icons mark the goal position.

distance between the current cell and the goal cell, i.e. the sum of x- and y-distance. The property we are interested in can be specified in HyperLTL as follows.

$$\exists\pi_1.\forall\pi_2.\square(h(s_{\pi_1}) \leq h(s_{\pi_2})) \tag{7.1}$$

The LTL-part makes use of considerable amounts of syntactic sugar. It expands to an expression that constraints the states of π_2 based on the state of π_1. Intuitively, it encodes the evaluation of the heuristic function explicitly for each position in the grid. For the left problem instance given in Figure 7.5, it expands to

$$\begin{aligned}\exists\pi_1.\forall\pi_2.\square((&\mathtt{x0y0}_{\pi_1} \rightarrow \mathtt{x0y0}_{\pi_2})\\ &\wedge(\mathtt{x1y0}_{\pi_1} \rightarrow(\mathtt{x0y0}_{\pi_2} \vee \mathtt{x1y0}_{\pi_2} \vee \mathtt{x0y1}_{\pi_2}))\\ &\wedge(\mathtt{x0y1}_{\pi_1} \rightarrow(\mathtt{x0y0}_{\pi_2} \vee \mathtt{x1y0}_{\pi_2} \vee \mathtt{x0y1}_{\pi_2}))\\ &\wedge(\mathtt{x1y1}_{\pi_1} \rightarrow(\mathtt{x0y0}_{\pi_2} \vee \mathtt{x1y0}_{\pi_2} \vee \mathtt{x0y1}_{\pi_2} \vee \mathtt{x1y1}_{\pi_2})))\end{aligned}$$

where $\mathtt{xiyj}_\pi$ is a shortcut for the respective predicate. This formula holds, we find a proof at bound 3. The right instance given in Figure 7.5 including the dashed wall is an example where the property does not hold. The (unique) optimal plan in this case visits `x0y2` after three steps. But within the same number of steps we could reach position `x2y1` as well, which is closer to the goal according to our heuristic function. Removing the dashed wall leads to a system satisfying the property. The bound at which a witness occurs is 11.

It turns out that in this domain, property 7.1 can always be proven (or disproved) within the reachability diameter (plus one technically such that we observe a loop). This is because the loop size of ς_2 is always one and we both quantified traces refer to the same system. Actually this holds for all sample tasks we have chosen so far, in this sense our choice was somewhat biased. The next task will be different in this regard. Before moving there, we elaborate a bit on the scalability for the current task.

The complexity depends arguably mostly on the reachability diameter. We construct a parametrised problem of based on quadratic grids where the reachability diameter corresponds roughly to the the side length. Our initial observation is that model checking of a 8×8 grid takes already around five minutes. It turns out that the bottleneck is currently the system encoding, namely the translation from PDDL to BTOR2. Our general-purpose translation does not perform any optimisations so far. For example, the predicates capturing the transitions (analogous to `road` in Figure 7.2) are encoded as states even though they are actually constants. A tailor-made translation for this task leads to a significant runtime improvement such that we can handle a 20×20 grid in about the same time as the 8×8 previously. Now, we are in the same order of magnitude as reported by Hsu et al. [10] in a similar planning task (i.e. grid world and nearly equal unrolling bound).

Note that applying a similar optimisation in the previous two settings does not help since the bottleneck is elsewhere, namely at the loop computation.

Collision Avoidance

So far, all our infinite witnesses incorporate a trivial loop, namely they end in a single state that repeats infinitely often. At some point, we reach a goal and it suffices to perform stutter steps afterwards. This setting is not representative in general and does not fully exploit the capabilities of an unbounded globally operator.

Our formula fragment is more powerful in the sense that it is able to verify continuous planning tasks, where the goal is some invariant that must hold in each step. This is only sensible if the state of the existential path must actually change infinitely often in order to achieve this goal, otherwise we are again in the previous case.

Specifying such formulas does not make much sense as long as a stutter step is included in the universally quantified system. In this case, ς_2 always ends with the set of reachable states and loop size one. The stutter step requires that from some point onwards, the invariant must hold for any state reachable by an universal path. This implies that there is no need to proceed for the existential path.

We already mentioned before that our approach is potentially more efficient if the universal path is further restricted. Our next task targets exactly this insight. We remain in the grid-world setting. Each path variable defines an individual agent that moves on this grid, so both variables may refer to different systems. The universal agent moves on a fixed set of routes, but still there is some uncertainty which path exactly it takes. On the other hand, the existential agent may move freely across the grid for now, but in principle we can impose some further restrictions here as well. The invariant is that both

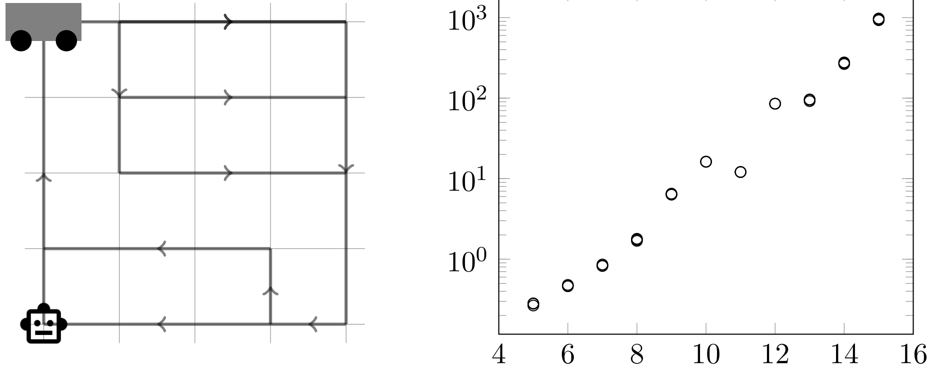

Figure 7.6: Collision avoidance task. Left: The robot carriage moves along the thick lines while the robot may move freely. The robot must commute between opposite corners. Right: Runtime (in seconds) compared to the grid size, i.e. side length. The left task has size 5.

agents never occupy one location at the same time, formally

$$\exists \pi_1 . \forall \pi_2 . \Box (\bigwedge_{i,j} \neg (\mathtt{xiyj}_{\pi_1} \land \mathtt{xiyj}_{\pi_2})). \tag{7.2}$$

As a rough motivation of this setting, you can think of the universal agent as a railway carriage moving on a rail track in a factory while some new agent must find its own way while avoiding any collisions. A sample task is given in Figure 7.6. The thick lines capture the paths the carriage may take. An additional constraint we impose now is that an agent must not stand still, i.e. we do not include a stutter action. We create an encoding tailored specifically to this domain since it is not easy to achieve this in general as mentioned before.

For the task given in Figure 7.6, we require that the existential agent must always move back and forth between its initial position and the top right corner such that it actually crosses the path of the carriage. This is ensured by constraining the location of the existential agent at these points based on the known location of the carriage. The loop of ς_2 is ranges from 0 to 16 in this scenario. The incremental algorithm also finds a witness within 16 steps which corresponds to the number of steps it takes the carriage to complete a full cycle. Note that at least the trail from down left to top right is uniquely defined. The existential agent must first go straight up and turn right only at the end in order to avoid a collision. We are able to verify that this is indeed the path provided as a solution by Z3.

In this setting, the algorithm scales again quite good to larger problems. This is essentially due to the fact that the computation of the loop in ς_2 is easy. The complexity of the remaining computation, namely the evaluation of the $\exists\forall$ SMT problem, is comparable to the incomplete bounded model checking algorithm from [10]. Concrete, we can scale the task to a 10×10 grid while keeping the runtime as low as a few seconds. This corresponds to a bound of 36. But from then onwards, the runtime increases noticeable. For a 15×15 grid, it takes about 15 minutes to obtain a witness within 56 steps. A detailed overview of the runtime compared to the grid size is given in Figure 7.6. It grows exponential with the side length. We repeated the experiments three times, but deviations in the runtime are barely noticeable on a logarithmic scale.

7.3 $\forall\exists$ + LTL Invariant

Now we discuss the implementation of the algorithm described in Section 6.5. It appears that several interesting properties lie in the respective HyperLTL fragment, presumably more than in the fragment discussed in the previous section. For instance, language containment mentioned in Example 3.3 is such a property. It is widely used in automaton-based model checking [49]. Also the verification of symmetry properties in the Bakery algorithm [50] serving as benchmarks to other HyperLTL model checkers belongs to this fragment [11, 10, 18]. Another example that was particularly used in the evaluation of HyperQube is linearizability [10, 18].

Linearizability

Linearizability is some notion of correctness for concurrent data types [51]. Basically it requires that any concurrent program execution may be simulated by another sequential execution of the same program. How to express linearizability as a hyperproperty is discussed in [52]. It suffices to ensure that every concurrent program execution is observationally equivalent to some sequential execution. Here, observational equivalence means that all method invocations as well as all returned values of the data type are identical. Translated to HyperLTL, we obtain

$$\forall\pi_1.\exists\pi_2.\square\left(\bigwedge_{obs} obs_{\pi_1} \leftrightarrow obs_{\pi_2}\right) \tag{7.3}$$

where π_1 is a trace from the concurrent program whereas π_2 is a trace from the respective sequential program. *obs* is an atomic proposition representing an observable event.

```
atomic cas(int *p, int old, int new) -> bool {
        if (*p != old) {
                return false;
        }
        *p = new;
        return true;
}
```

Listing 7.1: Compare-and-swap instruction

The authors of [10] use their BMC algorithm for HyperLTL in order to detect a linearizability violation in a particular concurrent data structure based on the Snark algorithm [53]. But their approach is unable to verify that the linearizability property is actually satisfied in a corrected version of the algorithm [54]. This is something that our algorithm can potentially achieve.

Since proving linearizability is far more difficult than finding a single bug, we consider a simpler data structure in a first attempt. One of the arguably most simple concurrent data structures is a counter, implemented using an atomic compare-and-swap instruction (see Listing 7.1) which is widely supported at hardware-level. Such an instruction prevents unexpected computations caused by two processes interfering with each other. The counter stores the current value as a bit-vector and supports only an increment operation which returns the new value of the counter after incrementing. A single instance of the counter is accessed by two concurrent processes. The aim is to prove that such a counter satisfies the linearizability property 7.3.

It turns out that even such a seemingly small property is a considerable challenge for our algorithm in practice. Z3 in the default configuration is unable to find a proof within hours, no matter how small we chose the counter size. The algorithm is stuck at a small two digit bound. In order to understand why this is the case, we examine yet another task in more detail, chosen for illustrative purposes.

Language Containment

Our aim is to verify whether certain BTOR2 operators can be simulated by other ones. The motivation is that BTOR2 supports a bunch of operators and apparently several of them are redundant in the sense that they can be eliminated by substitution with another operator. For now, we consider the left bit-shift operator as well as the multiplication operator. We consider the system depicted in Figure 7.1.1 for the former operator. For the latter one, we define an analogous system. The HyperLTL property of interest is actually a language containment property.

$$\forall \pi_{<<} . \exists \pi_{\star} . \square \left(s_{\pi_{<<}} \leftrightarrow s_{\pi_{\star}} \right) . \tag{7.4}$$

Admittedly, the motivation for this example is somewhat contrived, but it serves its purpose of being simple and parametrisable. We may customise both the input size and the state size of either system by altering the respective bit-vector size.

As a brief remark, note that the above formula contains a bit-vector state at a place where we would normally assume an atomic proposition. Formally, we can resolve this by introducing some atomic propositions, each representing a single bit of the state. But since we are encoding the formula into SMT, this is not necessary in practice. As a matter of fact, we may allow any kind of bit-vector arithmetic in the HyperLTL formula without much effort, as long as they are supported by SMT. In practice this is convenient since it allows for more concise formulas.

Before discussing the above task in more detail, it is worth pointing out a crucial characteristics of our algorithm. Remember that we aim to determine a witness function. Thus, we are existentially quantifying over a bit-vector function at the outermost level of the encoded SMT formula. Fortunately, bit-vector functions are natively supported by Z3. A reduction to QBF would be more cumbersome since functions are not a supported by QBF. Apart from involving functions, the encoding is mostly straightforward and follows Definition 6.2. The only technical difference is that we do not define only one function, but instead split it up into one function per input of the $\exists$-path such that each input can be replaced by a function application.

Returning to our actual problem statement again, it is evident that the validity of formula 7.4 depends on input- and state-size of both programs. For instance if we have state size four and input size one for both systems (just as in Figure 7.1.1) the property does not hold. A counterexample, i.e. a proof for the negated formula $\exists\pi_{<<}.\forall\pi_{*}.\Diamond(s_{\pi_{<<}} \not\leftrightarrow s_{\pi_{*}})$, can be easily discovered within one step using the approach discussed in Section 4.2. If we chose the input one for $\pi_{<<}$ in the first step, then $s_{\pi_{<<}} = \texttt{b0010}$ holds afterwards. But multiplying `b0001` by zero or one does not result in this value.

We are able to fix this issue by allowing a 2-bit input for the multiplication program instead. Now our algorithm finds a witness within exactly five steps. This is actually the lowest possible bound that is permitted solely by the simple path constraint. A path prefix for π_{*} consisting of only four steps does not necessarily contain a loop. Four is also the reachability recurrence diameter in this case, or equivalently the state size plus one.

It is not hard to come up with a proper witness function manually in this case. It suffices that all individual inputs are mapped such that at each point in time, a 0-bit-shift is simulated by multiplication by one and a 1-bit-shift is simulated by multiplication by two. This is essentially discovered by Z3 as well, except that the function representation is kind of bloated. Furthermore, the solution is not unique for the last two steps due to overflow. Both operations return zero by definition if overflow occurs.

	4-1-2	4-2-4	5-2-4	8-1-2	8-1-8	9-1-2
time (s)	< 1	110	18000	55	75	550
bound	5	5	6	9	9	10

Figure 7.7: Results for model checking property 7.4. The first number in the name represents the size of the state bit-vector. The second and the third one are the input size for $\pi_{<<}$ respectively $\pi_{\star}$.

The SMT problem for this task is solved by Z3 in a fraction of a second. This encourages us to increase the problem size. Doubling the state size to eight bits already increases the runtime to about a minute while the bound raises as well. Doubling the input size instead increases the runtime even more, but the bound does not change at all. Increasing both at the same time takes considerably more time. Z3 discovers a proof only after hours. In all cases, we can clearly observe that solving the SMT problem in the final step (which is actually satisfiable) accounts for nearly all of the runtime. The results are summarized in Figure 7.7.

Now we want to investigate why solving such a apparently small problem is actually that hard. In general, the most significant source of complexity in our use case is handling the quantifiers. Remember that our final SMT problem is an $\exists^*\forall^*$ formula, but this still coincides with our experiments in Section 7.2. What complicates this problem even more is that we are existentially quantifying over bit-vector functions now. This is an integral part of our algorithm since we are looking for a witness function which is able to resolve the existential quantifier.

To gain more insight in the complexity of the SMT problem, we examine the variables occurring in the SMT formula. At bound k, we have $2 \cdot k$ bits for the loop variables as well as $2 \cdot k \cdot |i_{\pi_\star}|$ bits for the inputs of the $\forall$-paths, where $|i_{\pi_\star}|$ is the size of the respective input bit-vector. Those are all universally quantified variables, which is not too bad. Additionally, we are searching for the witness function of the shape $k \cdot |i_{\pi_{<<}}| \to k \cdot |i_{\pi_\star}|$. Even though this is the only existentially quantified variable, we must acknowledge that the number of Boolean functions is doubly exponential in the number of arguments. Thus, most certainly the argument size of the witness function is precisely the bottleneck in our case.

This is indeed what the runtime results given in Figure 7.7 suggest. Increasing only the input size of $\pi_\star$ has negligible effect on the runtime in this context. On the other hand, increasing the input size of $\pi_{<<}$ has a drastic impact on the runtime since is linearly related to the argument size of the witness function. The state size affects the input size as well since it roughly corresponds to the final bound k, which explains why choosing a large state size is infeasible likewise.

Now we can understand why the linearizability example from the previous section is infeasible. The input size is not the issue, our encoding requires only one bit for the universal system and two bits for the existential system which is comparable to the tasks considered here. But the bound grows larger, concretely we are stuck at bound 29. Similar to the tasks here, almost all of the runtime is spend at the final bound. This suggests that the higher bound is caused by a larger state space rather than by a bug in our modelling.

7.4 Discussion

Overall, we can observe that the search for infinite witnesses is potentially quite expensive. Nevertheless, the results seem to be reasonable.

In the $\exists\forall$ case, the bounds we obtained are all related to the reachability diameter or variations thereof, similar to the completeness threshold of LTL invariants. As long as the universal path does not entail thousands of reachable states, the algorithm is feasible. Finding infinite witnesses in often not harder than finding finite ones in this case. On the other hand, the approach is infeasible if the state space of the universal path is huge. But as pointed out above, sensible properties usually can be rephrased to properties having a finite witness in this case. Including a stutter action into the universal system is the most extreme example in this regard.

For the $\forall\exists$ fragment, we did not manage to find a feasible procedure yielding infinite witnesses. This is somewhat unfortunate since many practically relevant properties belong to this fragment. The approach of determining a finite witness function via SMT solving may be applicable to many problems, but it turns out that this quickly overshoots the capabilities of modern SMT solvers.

Chapter 8

Conclusion

8.1 Summary

We started with the intention to extend existing HyperLTL BMC with completeness results by employing techniques well known from LTL BMC. We pointed out that such an extension is easy as long as no quantifier alternation is involved, based on the system composition proposed in [11]. But as soon as quantifier alternation occurs, we are unable to lift the techniques known from complete LTL BMC in a straightforward manner. The main reason is that loops do not allow us to draw conclusions about the infinite behaviour analogously since it does not suffice to argue about a single path respectively loop.

Against this background, we simplified the problem by focussing on fragments of HyperLTL including one quantifier alternation and an LTL invariant. As a general-purpose strategy to achieve completeness results, we mentioned how a completeness threshold may be derived from automaton-based HyperLTL model checking. But we realised quickly that bounds obtained via this method are either far too large or too hard to compute. Thus, we developed other individual model checking algorithms for both fragments.

For our $\exists\forall$ fragment, we were able to derive a completeness threshold via an alternative method. Based on loop detection in the universal path, we are able to determine at which point all states reachable in the composed system are guaranteed to have been visited. While the resulting bound is still quite large in practice, we are able to reduce it by fixing the existential path. This leads to an incremental algorithm that terminates as soon as a witness involving a fixed choice for the existential path has been found.

For the other fragment with switched quantification, it is more complicated to obtain a witness. A general witness in this case may be perceived as a function resolving the existential quantifier, mapping the chosen infinite universal path to an appropriate existential path. Our overall approach was to simulate such a function based on a bounded variant. While this is possible in many cases, it introduces quantification over a function variable to the encoded formula.

It turned out that even those two restricted fragments are hard to verify in practice. In the $\exists\forall$ fragment, we were able to apply our algorithm successfully to several sample tasks with decent performance. A vital premise is that the state space of the universal path is somewhat restricted to, say, a few thousand states. The $\forall\exists$ fragment actually contains several interesting properties. Unfortunately, we observed that even fairly small examples are largely infeasible for our algorithm. We noticed that this is mostly caused by the fact that the underlying SMT problem includes unbound bit-vector functions.

8.2 Future Work

Based on our investigations, we conjecture that model checking HyperLTL in general is a fairly challenging task if we aim for a completeness result. When it comes to the verification of hyperproperties in the real-world, it does not help to have a complete algorithm if it is infeasible for most non-trivial tasks. Thus, the most promising path seems to be imposing some restrictions on the specification.

Our attempts have been limited on restricting the formula shape so far. Another approach simplifying the task is to require all systems to have certain characteristics. This idea is to some extent pursued in the original work proposing HyperLTL BMC [10]. The halting semantics introduced there allows to draw more conclusion if all states end up in a self-loop. Potentially it is possible to come up with some less restrictive requirements targeting the system structure.

On the other hand, we can try to be less restrictive regarding the formula fragment. We may allow more than two quantifies, as long as they are not alternating more than once. This is a straightforward extension of our existing implementation that basically requires to employ the system composition idea. Allowing more quantifier alternation is rather not top priority since for most meaningful properties one alternation is sufficient [5]. Extending the algorithms to formulas consisting of LTL formulas beyond invariants is more desirable.

Allowing next operators as well is potentially realisable without much effort. They may be eliminated by introducing new states to the system, serving as delays. Beyond this, an extension to further safety properties suggests itself. But it is not obvious how to argue that we may extrapolate from bounded behaviour to unbounded behaviour for general safety properties. So far, we are relying on fact that it suffices to evaluate a propositional formula at each point in time individually for invariants.

Keeping in mind that even our existing algorithms become infeasible quite quickly, simply increasing the scope of the algorithms is possibly of no value in practice. Especially for our $\forall\exists$ fragment, it would be nice to have some more feasible procedure instead. We detected several highly relevant properties in this fragment, even if we solely consider invariants in the LTL body. To some extend, a high runtime is surely the price we have to pay for the extended expressiveness. But it is an open question whether we can abstain from computing a witness function explicitly via SMT solving, which is the bottleneck of our current algorithm.

Another issue that we did not discuss at all is to what extent bounded model checking may be applicable to HyperCTL*, which differs from HyperLTL only by allowing quantifiers to occur at any place inside the formula. While there exists work about bounded model checking for the branching-time logic CTL* (the corresponding non-hyper logic), an analogous procedure for HyperCTL* has not been proposed yet.

Index

BTOR2, 50

boolean formula, 19
bound (k), 21
bounded model checking (BMC), 21
Büchi automaton ($\mathcal{A}$), 9

Cartesian product ($\times$), 7
completeness threshold (c), 27
counterexample, 21

execution (α, δ^n), 8, 9

generalized noninterference (GNI), 2

HyperLTL
- bounded semantics ($\models^k$, $\dot{\models}^k$), 17
- semantics ($\models$), 16
- syntax, 15

hyperproperty, 11

invariant, 12

language containment, 16
linear-temporal logic (LTL)
- bounded semantics ($\models^k$, $\dot{\models}^k$), 13
- semantics ($\models$), 12
- syntax, 12

linearizability, 62

negation normal form (NNF), 13

path (σ, ς), 8, 9

PDDL, 52
planning, 51
power set ($\mathcal{P}$), 7
prefix, 9
prenex normal form, 18
projection ($|_A$), 10
propositional formula, 12

quantified boolean formula (QBF)
- semantics ($\models$), 18
- syntax, 17

quantifier (Q), 18
quantifier alternation, 15

reachability diameter (rd), 28
reachability recurrence diameter (rrd), 29
reachable ($Reach$), 9
run, 10

safety property, 10
satisfiability modulo theories (SMT), 19
sequence, 7
set, 7
successor ($Post$), 9
system ($S, s_0, Act, \delta, AP, L$), 8
system environment Γ, 15

temporal logic, 12
trace (t, TR, Tr, $Traces$), 9
trace assignment (Π), 15
trace property, 11
trace variable (π), 15

variable assignment (v), 18

word (w), 10

zip (zip), 7

Bibliography

[1] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pp. 51–65, IEEE Computer Society, 2008.

[2] M. N. Rabe, *A temporal logic approach to information-flow control.* PhD thesis, 2016.

[3] Y. Wang, S. Nalluri, and M. Pajic, "Hyperproperties for robotics: Planning via hyperltl," in *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*, pp. 8462–8468, IEEE, 2020.

[4] B. Finkbeiner, "Model checking algorithms for hyperproperties (invited paper)," in *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings* (F. Henglein, S. Shoham, and Y. Vizel, eds.), vol. 12597 of *Lecture Notes in Computer Science*, pp. 3–16, Springer, 2021.

[5] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal logics for hyperproperties," in *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings* (M. Abadi and S. Kremer, eds.), vol. 8414 of *Lecture Notes in Computer Science*, pp. 265–284, Springer, 2014.

[6] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pp. 46–57, IEEE Computer Society, 1977.

[7] E. A. Emerson and J. Y. Halpern, ""sometimes" and "not never" revisited: on branching versus linear time temporal logic," *J. ACM*, vol. 33, no. 1, pp. 151–178, 1986.

[8] D. McCullough, "Noninterference and the composability of security properties," in *Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 18-21, 1988*, pp. 177–186, IEEE Computer Society, 1988.

[9] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings* (R. Cleaveland, ed.), vol. 1579 of *Lecture Notes in Computer Science*, pp. 193–207, Springer, 1999.

[10] T. Hsu, C. Sánchez, and B. Bonakdarpour, "Bounded model checking for hyperproperties," in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I* (J. F. Groote and K. G. Larsen, eds.), vol. 12651 of *Lecture Notes in Computer Science*, pp. 94–112, Springer, 2021.

[11] B. Finkbeiner, M. N. Rabe, and C. Sánchez, "Algorithms for model checking hyperltl and hyperctl*," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I* (D. Kroening and C. S. Pasareanu, eds.), vol. 9206 of *Lecture Notes in Computer Science*, pp. 30–48, Springer, 2015.

[12] B. Finkbeiner, C. Müller, H. Seidl, and E. Zalinescu, "Verifying security policies in multi-agent workflows with loops," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 633–645, ACM, 2017.

[13] B. Finkbeiner, C. Hahn, and H. Torfah, "Model checking quantitative hyperproperties," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I* (H. Chockler and G. Weissenbacher, eds.), vol. 10981 of *Lecture Notes in Computer Science*, pp. 144–163, Springer, 2018.

[14] N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup, "Verifying hyperliveness," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (I. Dillig and S. Tasiran, eds.), vol. 11561 of *Lecture Notes in Computer Science*, pp. 121–139, Springer, 2019.

[15] M. Y. Vardi and P. Wolper, "Reasoning about infinite computations," *Inf. Comput.*, vol. 115, no. 1, pp. 1–37, 1994.

[16] S. Safra, "On the complexity of omega-automata," in *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pp. 319–327, IEEE Computer Society, 1988.

[17] M. N. Rabe and N. Coenen, "MCHyper." `www.react.uni-saarland.de/tools/mchyper`, 2022.

[18] T. Hsu, B. Bonakdarpour, and C. Sánchez, "Hyperqube: A qbf-based bounded model checker for hyperproperties," *CoRR*, vol. abs/2109.12989, 2021.

[19] T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila, "Simple bounded LTL model checking," in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings* (A. J. Hu and A. K. Martin, eds.), vol. 3312 of *Lecture Notes in Computer Science*, pp. 186–200, Springer, 2004.

[20] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[21] M. Cadoli, A. Giovanardi, and M. Schaerf, "An algorithm to evaluate quantified boolean formulae," in *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA* (J. Mostow and C. Rich, eds.), pp. 262–267, AAAI Press / The MIT Press, 1998.

[22] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, pp. 117–148, 2003.

[23] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, "SAT competition 2020," *Artif. Intell.*, vol. 301, p. 103572, 2021.

[24] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)." `www.SMT-LIB.org`, 2022.

[25] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[26] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 171–177, Springer, 2011.

[27] L. Tentrup, "CAQE and quabs: Abstraction based QBF solvers," *J. Satisf. Boolean Model. Comput.*, vol. 11, no. 1, pp. 155–210, 2019.

[28] M. Helmert, "The fast downward planning system," *J. Artif. Intell. Res.*, vol. 26, pp. 191–246, 2006.

[29] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^20 states and beyond," in *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pp. 428–439, IEEE Computer Society, 1990.

[30] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman, "Completeness and complexity of bounded model checking," in *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings* (B. Steffen and G. Levi, eds.), vol. 2937 of *Lecture Notes in Computer Science*, pp. 85–96, Springer, 2004.

[31] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," in *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA* (H. R. Lewis, B. B. Simons, W. A. Burkhard, and L. H. Landweber, eds.), pp. 159–168, ACM, 1982.

[32] D. Kroening and O. Strichman, "Efficient computation of recurrence diameters," in *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings* (L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, eds.), vol. 2575 of *Lecture Notes in Computer Science*, pp. 298–309, Springer, 2003.

[33] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, and J. Worrell, "Linear completeness thresholds for bounded model checking," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 557–572, Springer, 2011.

[34] K. Heljanko, T. A. Junttila, and T. Latvala, "Incremental and complete bounded model checking for full PLTL," in *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings* (K. Etessami and S. K. Rajamani, eds.), vol. 3576 of *Lecture Notes in Computer Science*, pp. 98–111, Springer, 2005.

[35] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," in *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings* (N. Halbwachs and D. A. Peled, eds.), vol. 1633 of *Lecture Notes in Computer Science*, pp. 172–183, Springer, 1999.

[36] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)* (F. Moller and G. M. Birtwistle, eds.), vol. 1043 of *Lecture Notes in Computer Science*, pp. 238–266, Springer, 1995.

[37] K. Havelund and D. Peled, "Runtime verification: From propositional to first-order temporal logic," in *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings* (C. Colombo and M. Leucker, eds.), vol. 11237 of *Lecture Notes in Computer Science*, pp. 90–112, Springer, 2018.

[38] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," in *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings* (E. M. Clarke and R. P. Kurshan, eds.), vol. 531 of *Lecture Notes in Computer Science*, pp. 233–242, Springer, 1990.

[39] "Rust." `www.rust-lang.org`, 2022.

[40] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , btormc and boolector 3.0," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I* (H. Chockler and G. Weissenbacher, eds.), vol. 10981 of *Lecture Notes in Computer Science*, pp. 587–595, Springer, 2018.

[41] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, and K. Heljanko, "Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 135–172, 2014.

[42] A. Biere, T. van Dijk, and K. Heljanko, "Hardware model checking competition 2017," in *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017* (D. Stewart and G. Weissenbacher, eds.), p. 9, IEEE, 2017.

[43] M. Ghallab, D. S. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[44] J. Hoffmann, "Everything you always wanted to know about planning - (but were afraid to ask)," in *KI 2011: Advances in Artificial Intelligence, 34th Annual German Conference on AI, Berlin, Germany, October 4-7,2011. Proceedings* (J. Bach and S. Edelkamp, eds.), vol. 7006 of *Lecture Notes in Computer Science*, pp. 1–13, Springer, 2011.

[45] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL—The Planning Domain Definition Language," 1998.

[46] D. Long and M. Fox, "The 3rd international planning competition: Results and analysis," *J. Artif. Intell. Res.*, vol. 20, pp. 1–59, 2003.

[47] "Create an array with fixed size and initialize it." `https://stackoverflow.com/a/11069399/1`, 2012.

[48] "Btor2tools." `wwww.github.com/boolector/btor2tools`, 2022.

[49] J. Esparza, O. Kupferman, and M. Y. Vardi", "Automata-theoretic verification," in *Handbook of Automata Theory* (J.-É. Pin, ed.), vol. 2, ch. 38, pp. 1415–1456, European Mathematical Society, 2011.

[50] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[51] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[52] B. Bonakdarpour, C. Sánchez, and G. Schneider, “Monitoring hyperproperties by combining static analysis and runtime verification,” in *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II* (T. Margaria and B. Steffen, eds.), vol. 11245 of *Lecture Notes in Computer Science*, pp. 8–27, Springer, 2018.

[53] D. Detlefs, C. H. Flood, A. Garthwaite, P. A. Martin, N. Shavit, and G. L. S. Jr., “Even better dcas-based concurrent deques,” in *Distributed Computing, 14th International Conference, DISC 2000, Toledo, Spain, October 4-6, 2000, Proceedings* (M. Herlihy, ed.), vol. 1914 of *Lecture Notes in Computer Science*, pp. 59–73, Springer, 2000.

[54] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele, “Dcas is not a silver bullet for nonblocking algorithm design,” in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’04, (New York, NY, USA), p. 216–224, Association for Computing Machinery, 2004.