SAARLAND UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Bachelor's Thesis

HyperLTL Synthesis

Author
Philip Lukert

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Advisor
Christopher Hahn

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr. Sebastian Hack
15th February 2018

Eidesstattliche Erklärung

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

Statement in Lieu of an Oath

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

Einverständniserklärung

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

Declaration of Consent

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

Saarbrücken, 15th February, 2018

Abstract

Ensuring secure information flow in privacy critical systems is an important aspect in system design. An example of such an information flow policy is observational determinism, which ensures that two traces with the same observable input have the same observable output. The temporal logic HyperLTL is designed to specify such properties by extending LTL with explicit trace quantification. Therefore, a HyperLTL formula defines a set of sets of traces instead of a single set of traces. Current work focuses on monitoring and model checking of a HyperLTL formula on a given system. In this work, we study the synthesis problem of HyperLTL. Synthesis is to directly generate a system from a given specification with no need for manual implementation, monitoring or model checking. In general, the synthesis problem of HyperLTL is undecidable. In this thesis, we show that the synthesis problem of the existential fragment (using solely existential quantifiers) is equivalent to the satisfiability problem of HyperLTL and is, therefore, fully decidable. We also give a decidable sub-fragment of the universal fragment as the whole universal fragment turns out to be undecidable. Further, we give a undecidability proof for universal formulas in general. Furthermore, we proof the undecidability of the synthesis problem for formulas with a quantifier alternation from universal to existential by a reduction from Post's Correspondence Problem (PCP).

The results of this thesis have been included in the manuscript "Synthesizing Reactive Systems from Hyperproperties" with the co-authors Bernd Finkbeiner, Christopher Hahn, Marvin Stenger and Leander Tentrup. The manuscript is currently under review for publication.

Acknowledgements

First of all I thank my advisor Chris. He helped me a lot with this thesis and had always a free minute for me. Second I would like to thank my supervisor Prof. Finkbeiner who gave me this interesting and challenging topic and is my first reviewer. Many thanks also to my second reviewer Prof. Hack. I also would like to thank Jana and Lennart for proofreading my thesis additionally to Chris. And thanks for the interesting and inspiring discussions in the lab, Carsten and Marvin. I also thank Leander who gave me the idea to have a look on the deterministic existential fragment. Last but not least I would like to thank my parents Britta and Georg and my little sister Luisa.

Contents

Chapter 1

Introduction

There have been many security critical vulnerabilities such as Hartbleed [1], Spectre [13] or Meltdown [14] to just name a few. They are all bugs that gave access to a private resource enabling an information flow of sensitive data to the attacker. In information flow control, we would like to have no information flow from private data into the public domain. As an example information flow property, observational determinism [18, 20] specifies that the public output should only depend on the public input. Information flow properties are so called hyperproperties [6]. In contrast to usual trace properties, hyperproperties reason about sets of traces instead of single traces in isolation. This enables to reason about relations between two or more traces. Being able to relate multiple traces enables one to say "on two execution traces with the same public input, we want to have the same public output".

HyperLTL [7] is a temporal logic that formalizes these Hyperproperties by extending Linear-Time Temporal Logic (LTL) [15], which is only able to define trace properties. With universal ($\forall$) and existential ($\exists$) quantifiers, the logic is able to refer to either all traces in a trace set or to enforce a property to hold on at least one trace in the set. For example, observational determinism could be described as

$$\forall \pi, \pi'.\ (O_\pi \Leftrightarrow O_{\pi'})\ \mathsf{W}\ (I_\pi \not\Leftrightarrow I_{\pi'})$$

Here, we pick to two arbitrary traces π and π' and enforce that the outputs (O) have to be the same on both traces (weak-)until the inputs differ on the traces. So if the inputs are the same at all time, the outputs have to be the same at all time.

Synthesis is a technique to generate a system that is automatically correct according to a specification. This avoids human programming errors and speeds up the system design. Much work has already been done synthesizing hardware circuits because they are relatively simple in contrast to higher level programming languages. Due to the high interest, the annual Syntcomp [2], a competition on synthesis tools, established in 2014. However, the current focus is on synthesizing trace properties. This has the drawback that the produced systems cannot ensure information flow properties. As these systems are automatically constructed, it is difficult to change anything afterwards while preserving functionality. Especially

information flow properties are hard to implement. With synthesis from hyperproperties, a synthesized system could be designed to be correct and secure by construction.

The initially mentioned Meltdown is an example where information flow control - possibly in combination with synthesis - could have prevented a critical security vulnerability. Meltdown is a vulnerability in the processors of (almost) all computers. An essential part of that vulnerability is that a process can measure the time a request to the cache of the system takes to be answered. This request is answered fast if the data was already in the cache. Therefore, the process could measure whether an other process used the data recently. This is not intended and facilitates a whole series of attacks.

Many benchmark tests of the Syntcomp specify variants of an arbiter circuit. An arbiter is a circuit that manages access of different processes to a shared resource while ensuring mutual exclusion. It gets n request signals of different processes as input and outputs n grants stating whether the arbiter gives the access to process i. The arbiter has to take care to give access to at most one processor at a time and to eventually give a grant to each process that sent a request.

In such an arbiter we could have information flow, too. A process could gather information about the other requests by just looking at its own grant. The problem is that such information leakage cannot be prohibited by an LTL specification. We therefore need a Hyperproperty. Using observational determinism, we can enforce that the grant signal of a process depends only on its own request signal. With this specification, we prevent information leakage of the other requests.

We can divide HyperLTL into fragments by analyzing the used quantifiers. In HyperLTL quantifiers are only in the quantifier prefix allowed, i.e., no quantifiers occur after a temporal or propositional operator. We denote with the fragments $\forall$* and $\exists$* the class of formulas in which we use solely the respective quantifier. We also expand this notation by arbitrary combinations, e.g., $\forall$*$\exists$* means that the quantifier prefix of a formula inside this fragment contains first only $\forall$'s and then only $\exists$'s.

In this thesis, we determine fragments of HyperLTL for which the synthesis problem is decidable. We especially have a look at the $\exists$*-fragment and the $\forall$* fragment. The $\exists$* fragment turns out to be fully decidable and we give a sub-fragment of $\forall$* that is decidable, too. For the $\exists$* fragment, we show that the synthesis problem is equivalent to the satisfiability problem up to determinism. Satisfiability is already shown to be decidable for the $\exists$* fragment as well as the $\exists$*$\forall$* fragment [9]. Note that we need the $\forall$ quantifiers to specify the determinism.

The distributed synthesis problem for LTL [10] is realizing an LTL specification in a restricted architecture. Figure 1.1 shows such an architecture that restricts the output c to depend only on a and d on b. This property can again be specified with observational determinism in the $\forall$* fragment.

It turns out that there are some architectures, e.g., the one in Figure 1.1, in which

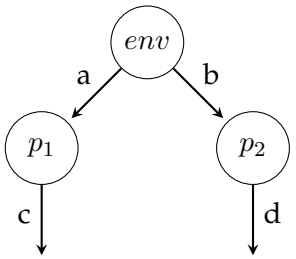

Figure 1.1: An architecture of two processes that specifies process p_1 to produce c from a and p_2 to produce d from b. env is a special environment process that provides the input variables.

the LTL synthesis problem is not decidable. As we can encode these architectures in the ∀* fragment, this fragment is in undecidable in general. In the distributed synthesis setting, there are architectures that are still decidable. These architectures are, in some way, linear[1]. They are as expressive as the fully decidable Coordination Logic [11]. In this thesis, we introduce an interesting sub-fragment of ∀* that can be reduced to Coordination logic and is, therefore, decidable.

Finally, we provide a undecidability proof of ∀*∃* (first only universal and then only existential quantifiers). We reduce Post's Correspondence Problem (PCP) to a HyperLTL formula with one ∀ quantifier followed by one ∃ quantifier. As PCP is undecidable, the synthesis problem from HyperLTL formulas with at least one quantifier alternation from ∀ to ∃ is undecidable.

We structure the thesis as follows. In Chapter 2, the required preliminaries are introduced. In Chapter 3, we analyze the different fragments. We will start by having a look at the ∃* fragment and differentiating between deterministic and nondeterministic synthesis and continue with the ∀* fragment. There, we first give a construction to reduce ∀* formulas to $\forall^1$ formulas (only one single ∀ quantifier). That procedure can be used to reduce the quantifier amount of a HyperLTL formula, if applicable. We also use it to define the so called CL-fragment of HyperLTL, which is a fragment that can be reduced to Coordination Logic. In the last section of the fourth chapter, we proof the undecidability of ∃*∀* formulas. In Chapter 4, we give a short overview over related work and in Chapter 5, we summarize our results and present some ideas for future work.

[1] I.e., they are architectures that contain no information forks [10].

Chapter 2

Preliminaries

Trace properties reason about infinite traces that represent executions of a system. In model checking, one ensures that all executions, i.e., traces of a system fulfill the trace property in isolation. In contrast to that, hyperproperties are properties that reason about sets of traces. Therefore it is able to relate multiple traces to each other. This increases expressivity enormously.

In the following sections, we give introductions into the logics LTL and HyperLTL that formalize trace properties and, respectively, hyperproperties.

2.1 LTL

LTL [15] is a logic that extends propositional logic with temporal operators which allows us to reason about traces of atomic propositions. Therefore, LTL defines trace properties. The two main additions are the next-operator $\bigcirc$, that refers to the trace with the first element skipped and the until-operator $\varphi \mathsf{U} \psi$, which states that formula φ holds until ψ holds. From the until-operator, we can derive some other operators, e.g., $\Box \varphi$, that states that φ always holds. With these formulas we can describe, for example, a request-grant property of an arbiter $\Box(request \Rightarrow \bigcirc grant)$: "every request is answered with a grant in the next step". The syntax is defined as follows.

Definition 2.1 (LTL Syntax)

$$\varphi \coloneqq a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \ \mathit{U} \ \varphi \mid \Diamond \varphi \mid \Box \varphi \mid \varphi \ \mathit{W} \ \varphi$$

where $a \in AP$ is an atomic proposition of the set AP. We can derive the operators finally: $\Diamond \varphi \coloneqq true \ \mathsf{U} \ \varphi$ ("there is a time step in which φ holds"), globally: $\Box \varphi \coloneqq \neg \Diamond \neg \varphi$ ("φ holds always"), and weak-until: $\varphi \ \mathsf{W} \ \psi \coloneqq (\varphi \ \mathsf{U} \ \psi) \vee \Box \varphi$ (like U with no need to fulfill φ at all). The other boolean operators $\wedge, \Rightarrow, \Leftrightarrow$ are derived as usual. The semantics are defined over traces $t \in (2^{AP})^\omega$. $t[n]$ denotes the element at position n while $t[n, \infty]$ denotes the trace with the first n elements cut off. We define $\models$ inductively by

Definition 2.2 (LTL Semantics)

$$
\begin{aligned}
t \models a &\quad\Leftrightarrow a \in t[0] \\
t \models \neg\varphi &\quad\Leftrightarrow t \not\models \varphi \\
t \models \varphi \vee \psi &\quad\Leftrightarrow t \models \varphi \textit{ or } t \models \psi \\
t \models \bigcirc \varphi &\quad\Leftrightarrow t[1,\infty] \models \varphi \\
t \models \varphi \ U \ \psi &\quad\Leftrightarrow \exists n \in \mathbb{N}.\ t[n,\infty] \models \psi \textit{ and } \forall k < n.\ t[k,\infty] \models \varphi
\end{aligned}
$$

Two formulas φ and ψ are called equivalent $\varphi \equiv \psi$ iff for all traces t holds $t \models \varphi$ iff $t \models \psi$. φ is called satisfiable iff there exists a trace t with $t \models \varphi$.

2.2 LTL Synthesis

For the synthesis problem, the atomic propositions are divided into input and output $AP = I \dot\cup O$. The result of the synthesis problem is a strategy-tree that maps an input sequence to an output $(2^I)^+ \mapsto (2^O)$. Intuitively, this tree represents the reaction (output) of a system to any possible sequence of inputs. Note that the strategy-tree maps from **non**-empty input sequences to an output. The traces of a strategy-tree t are defined by $traces(t) = \{(i_0 \cup o_0), (i_1 \cup o_1), \dots \mid i_n \in I, o_n = t(i_0, \dots, i_n) \text{ for all } n \in \mathbb{N}\}$.

Definition 2.3 (LTL Synthesis)
A strategy-tree t realizes an LTL formula φ iff $traces(t) \models \varphi$

2.3 HyperLTL

HyperLTL [7] extends LTL by a quantifier prefix consisting of universal ($\forall$) and existential ($\exists$) quantification to refer to multiple traces. Therefore, HyperLTL reasons about sets of traces $T \subseteq (2^{AP})^\omega$ instead of single traces and, therefore, defines hyperproperties. Our running example, observational determinism is a property that takes advantage of this: $\forall\pi, \pi'.(o_\pi \Leftrightarrow o_{\pi'})\ \mathsf{W}\ (i_\pi \not\Leftrightarrow i_{\pi'})$ states that the outputs of all traces behave deterministically, i.e., "for all traces the outputs are the same until the inputs differ". Note that this trivially holds for single traces. Formally, the syntax is defined as follows.

Definition 2.4 (HyperLTL Syntax)

$$
\begin{aligned}
\varphi &:= \forall\pi.\varphi \mid \exists\pi.\varphi \mid \phi \\
\phi &:= a_\pi \mid \neg\phi \mid \phi \wedge \phi \mid \bigcirc\phi \mid \phi \ U \ \phi \mid \Diamond\phi \mid \Box\phi \mid \phi \ W \ \phi
\end{aligned}
$$

where $a \in AP$ is an atomic proposition and $\pi \in V$ is a trace variable over the set of trace variables V. The atomic propositions a_π are always assigned to a trace π. The syntax is defined using an (initially empty) trace assignment $A \in V \to T$:

Definition 2.5 (HyperLTL Semantics)

$$
\begin{aligned}
T \models_A a_\pi &\Leftrightarrow a \in A(\pi)[0] \\
T \models_A \neg\varphi &\Leftrightarrow T \not\models_A \varphi \\
T \models_A \varphi \wedge \psi &\Leftrightarrow T \models_A \varphi \text{ and } T \models_A \psi \\
T \models_A \bigcirc \varphi &\Leftrightarrow T \models_{A[1,\infty]} \varphi \\
T \models_A \varphi \mathbin{U} \psi &\Leftrightarrow \exists n \in \mathbb{N}.\ T \models_{A[n,\infty]} \psi \text{ and } \forall k < n.\ T \models_{A[k,\infty]} \varphi \\
T \models_A \exists \pi.\ \varphi &\Leftrightarrow \exists t \in T.\ T \models_{A[\pi \mapsto t]} \varphi \\
T \models_A \forall \pi.\ \varphi &\Leftrightarrow \forall t \in T.\ T \models_{A[\pi \mapsto t]} \varphi
\end{aligned}
$$

where $A[n,\infty](\pi) \coloneqq A(\pi)[n,\infty]$ denotes the assignment where the first n elements are removed from each trace and $A[\pi \mapsto t]$ denotes the modification of A where π maps to t and everything else remains unchanged. A trace set T satisfies a HyperLTL formula φ denoted by $T \models \varphi$ iff $T \models_\varepsilon \varphi$ where ε is the empty assignment. Two formulas φ and ψ are called equivalent $\varphi \equiv \psi$ iff for all trace sets T holds $T \models \varphi$ iff $T \models \psi$. φ is called satisfiable iff there exists a set of strategy-trees T with $T \models \varphi$.

2.4 HyperLTL Synthesis

As in LTL synthesis, when discussing the synthesis problem from HyperLTL the atomic propositions are divided into input and output $AP = I \dot\cup O$ and the objective is to find a strategy-tree $(2^I)^+ \mapsto (2^O)$ that fulfills a given formula. The definition is the same as in LTL.

Definition 2.6 (HyperLTL Synthesis)
A strategy-tree t realizes a HyperLTL formula φ iff $traces(t) \models \varphi$.

2.4.1 Nondeterministic Synthesis

Note that HyperLTL formulas can enforce nondeterminism. For example, the formula $\exists \pi, \pi'.\ a_\pi \wedge \neg a_{\pi'}$ can only be satisfied with sets of at least two traces. Similarly, we can enforce formulas that can only be solved with a nondeterministic

strategy. For example, $\exists \pi, \pi'.\ \square(in_\pi \Leftrightarrow in_{\pi'}) \wedge \square(out_\pi \neq out_{\pi'})$ can be only realized by a system that acts differently on two runs with the same input. Such nondeterminism can be represented with sets of strategy-trees T. The traces of such a set are simply the union of all individual trace sets $traces(T) \coloneqq \bigcup_{t \in T}\ traces(t)$.

Definition 2.7 (Nondeterministic HyperLTL Synthesis)
A non-empty set of strategy-trees T ***nondeterministically*** *realizes a HyperLTL formula φ iff $traces(T) \models \varphi$.*

Chapter 3

HyperLTL Synthesis

It turns out that the synthesis problem from HyperLTL is undecidable in general. In this chapter, we will have a look on some fragments of HyperLTL and elaborate whether they are decidable. In particular, we will consider $\exists$*-formulas that use only $\exists$ quantifiers and $\forall$*-formulas (respectively). The $\exists$*-fragment turns out to be fully decidable. A reduction to the HyperLTL satisfiability problem which is decidable is given. The $\forall$*-fragment turns out to be partially decidable. We give the definition of a fragment which correlates with the fully decidable Coordination Logic [11]. We also prove that the rest of the fragment is not decidable.

The formal definition of the fragments is as follows. The $\forall^n$ fragment denotes all formulas in which the quantifier block consists of exactly n $\forall$-quantifiers and $\forall^* := \bigcup_{n \in \mathbb{N}^+} \forall^n$. Analogously for $\exists$.

3.1 $\exists$* Fragment

Synthesis from formulas in the $\exists$* fragment is possible. We divide this section into the analysis of the deterministic and the nondeterministic synthesis problem. For the nondeterministic synthesis it holds that a formula is realizable iff it is satisfiable. For the deterministic synthesis problem, we have to add observational determinism to the formula to achieve this equivalence between satisfiability and realizability.

3.1.1 (Deterministic) Synthesis from $\exists$*

In the deterministic case, we have to ensure that the traces of the satisfiability result do not have different outputs on the same input. That is because we are limited to a single tree in the deterministic synthesis problem. For example, $\exists \pi, \pi' . o_\pi \neq o_{\pi'}$ is satisfiable but not deterministically realizable with $I = \emptyset, O = o$. We enforce this by simply adding the observational determinism HyperLTL formula to the original formula.

Lemma 3.1 *Given an ∃* formula φ. The formula is realizable iff*

$$\psi \coloneqq \varphi \wedge \forall \pi, \pi'. \bigwedge_{o \in O} (o_\pi \Leftrightarrow o_{\pi'}) \ \mathsf{W} \bigvee_{i \in I} (i_\pi \not\Leftrightarrow i_{\pi'})$$

is satisfiable.

Proof

"⇒" Assume φ is realizable. Let t be the strategy-tree that realizes φ. Hence, $traces(t) \models \varphi$ holds. It also holds that there are no two traces with different output and same input (up to the point of input difference) because it is only one strategy-tree. So $traces(t) \models \psi$ holds, too. Therefore, ψ is satisfiable.

"⇐" Assume ψ is satisfiable. Let S be a set of traces that satisfies ψ. Construct a strategy-tree t as follows.

$$t \coloneqq in \mapsto \begin{cases} s[n] \cap O & \text{if for some } s \in S \text{ holds (1): } in \text{ is a prefix of } s|_I \\ & \text{with } n = |in| - 1 \\ \emptyset & \text{else} \end{cases}$$

Where $s|_I$ denotes the trace restricted to I, formally $n \mapsto s(n) \cap I$. The $|in|-1$ is needed because the 0th trace element should be the 1st tree element because there is no output for the empty input sequence. Note that if (1) holds for multiple s, then $s(n)$ is the same for all of them because of the added non-determinism-formula. We now have that $S \subseteq traces(t)$ because each s_i is in t. With $S \models \varphi$ because $\varphi \models \psi$ and φ being an existential formula, $traces(t) \models \varphi$. □

Theorem 3.2 *The synthesis problem for the ∃*-fragment of HyperLTL is decidable.*

Proof Note that ψ from Lemma 3.1 can be expressed in the ∃*∀*-fragment (first ∃ quantifiers and then only ∀ quantifiers in the quantifier block). As the satisfiability problem for the ∃*∀* fragment is decidable [9], the synthesis problem is decidable. □

Corollary 3.3 *Realizability of ∃* HyperLTL specifications is PSPACE-complete.*

Proof We gave a linear reduction to the satisfiability of the $\exists^*\forall^2$ fragment. As we have a bounded amount, i.e., two ∀ quantifiers, the satisfiability is in PSPACE. Therefore, the realizability is in PSPACE. The problem is also PSPACE-hard because LTL satisfiability that is equivalent to the $\exists^1$ fragment is PSPACE-hard. Therefore the problem is PSPACE-complete. □

3.1.2 Nondeterministic Synthesis from ∃*

In the nondeterministic case, the problem is even simpler. We now do not need to care about nondeterminism and can simply construct an arbitrary tree around each trace we get from the satisfiability solution. Therefore, there is a one-to-one correlation between satisfiability and synthesis.

Lemma 3.4 *A formula $\varphi \in \exists^*$ is nondeterministically realizable iff it is satisfiable.*

Proof

"⇒" Assume φ is nondeterministically realizable. Let T be the set of strategy-trees that realizes φ. It holds $traces(T) \models \varphi$ by definition of realizability. Therefore, φ is satisfiable.

"⇐" Assume φ is satisfiable. Let S be a set of traces that satisfies φ. For each trace $s_i \in S$ construct a set of strategy-trees T with elements t_i as follows.

$$t_i \coloneqq in \mapsto s_i[n] \cap O \text{ with } n = |in| - 1$$

Hence for each i, t_i contains the trace s_i because each trace of t_i satisfies the output-constraints of s_i and for all input-constraints of s_i there exists a path π_i in t_i that satisfies them. Therefore, $\pi_i = s_i$.

As $S \subseteq traces(T)$ and φ is an existential formula and $S \models \varphi$, $traces(T) \models \varphi$ holds what says that T realizes φ nondeterministically. □

Theorem 3.5
The nondeterministic synthesis problem for the ∃-fragment of HyperLTL is decidable.*

Proof As the satisfiability problem for the ∃* fragment is decidable [9], the nondeterministic synthesis problem is decidable by Lemma 3.4. □

3.2 ∀* Fragment

The ∀* fragment is a little bit more involved. It turns out that the synthesis problem from ∀* is in general undecidable (and therefore the synthesis from HyperLTL). But we give a sub-fragment that is still decidable. This sub-fragment correlates with Coordination Logic, i.e., we will find a reduction of this sub-fragment to Coordination Logic which is fully decidable.

In this Chapter, we will also have a short look on nondeterminism which turns out to be not interesting in ∀*. In ∀* we cannot enforce nondeterminism. We will also see some simplifications of formulas, i.e., we define a reduction from ∀* to $\forall^1$ to apply it whenever possible to simplify the formula.

3.2.1 Nondeterministic ∀*

In this section, we show that the nondeterministic synthesis from ∀* is the same as the deterministic synthesis from ∀*. Intuitively, this is because you cannot enforce nondeterminism in ∀*, i.e., from a nondeterministic solution (non-empty set of trees) all single trees are deterministic solutions because the traces of them are a subset of the traces of the tree-set.

Theorem 3.6
A ∀ formula φ is deterministically realizable iff it is nondeterministically realizable.*

Proof

"⇒" assume t realizes φ, i.e., $traces(t) \models \varphi$ and by definition $traces(\{t\}) \models \varphi$. Therefore $\{t\}$ nondet. realizes φ.

"⇐" assume T nondet. realizes φ, i.e., $traces(T) \models \varphi$. Note that for each ∀* formula φ it holds, that if a set X satisfies φ, also $Y \subseteq X$ satisfies φ. Pick one $t \in T \neq \emptyset$. Clearly $traces(t) \subseteq traces(T)$. Therefore $traces(t) \models \varphi$ and t realizes φ. □

Sometimes things get easier when enforcing determinism. We abbreviate our formula for observational determinism as follows.

$$D_{A \mapsto C} \coloneqq \bigwedge_{c_i \in C} (c_{i\pi} \Leftrightarrow c_{i\pi'}) \ \mathsf{W} \ \bigvee_{a_i \in A} (a_{i\pi} \not\Leftrightarrow a_{i\pi'})$$

denotes that C only depends on A. If we add $D_{I \mapsto O}$ to a formula, we enforce the outputs to depend only on the inputs, i.e., there cannot be a different output on the same input. This is the case when we have more than one tree. But it is not a restriction to the deterministic realizability problem because there we have only one tree.

Corollary 3.7 *A ∀* formula φ is realizable iff $\varphi \wedge D_{I \mapsto O}$ is realizable.*

3.2.2 Reduction from ∀* to $\forall^1$

In this chapter, we present a reduction from a formula with many universal quantifiers to a formula with only one quantifier. This is done by simply removing all but one quantifier and renaming the path variables. For example, $\forall\pi_1, \pi_2. \square a_{\pi_1} \vee \square a_{\pi_2}$ is reduced to its equivalent $\forall^1$ formula $\forall\pi. \square a_\pi \vee \square a_\pi$. However, this reduction is not always possible as $\forall\pi_1, \pi_2. \square(a_{\pi_1} \Leftrightarrow a_{\pi_2})$ is not equivalent to its reduction $\forall\pi. \square(a_\pi \Leftrightarrow a_\pi)$. Note that a HyperLTL formula with exactly one $\forall$ quantifier is an LTL property.

Definition 3.8 *Let $\varphi = \forall\pi_1, \pi_2, \ldots, \pi_n.\ \varphi'$ be a formula in ∀*. Define the reduced formula of φ by $reduce(\varphi) := \forall\pi.\ \varphi'[\pi_1 \mapsto \pi][\pi_2 \mapsto \pi] \ldots [\pi_n \mapsto \pi]$ where $\varphi'[\pi_i \mapsto \pi]$ denotes the renaming of all occurrences of π_i to π.*

Although the reduced term is not always equivalent to the original formula, we can use it as an indicator whether it is possible at all to express a ∀* formula with only one quantifier as we see in the following theorem.

Theorem 3.9 *It holds that $\varphi \equiv reduce(\varphi)$ or $\nexists \phi \in \forall^1.\ \phi \equiv \varphi$*

Proof Suppose there is some $\phi \in \forall^1$ with $\phi \equiv \varphi$. We will prove that $\phi \equiv reduce(\varphi)$. Let S be an arbitrary set of traces. Let $S' = \{\{s\} \mid s \in S\}$. Because $\phi \in \forall^1$, $S \models \phi$ is equivalent to $\forall s' \in S'.\ s' \models \phi$, what is by assumption equivalent to $\forall s' \in S'.\ s' \models \varphi$. Now, φ operates on singleton trace sets only. This means that all quantified paths have to be the same what yields that we can use the same path variable for all of them. So $\forall s' \in S'.\ s' \models \varphi \Leftrightarrow s' \models reduce(\varphi)$ what is again equivalent to $S \models reduce(\varphi)$. Because $\phi \equiv reduce(\varphi)$ and $\phi \equiv \varphi$ it holds that $\varphi \equiv reduce(\varphi)$. □

Reduction from $\forall^n$ to $\forall^{n-1}$

We can generalize the previous theorem as follows. Sometimes, the reduction to a single $\forall$ quantifier is impossible although it is possible to reduce the quantifier amount. Again, this is not always possible. Here we see a procedure to reduce the quantifier amount by one. We can do this iteratively until we find the minimal amount of quantifiers.

Let $\varphi = \forall\pi_1, \pi_2, \ldots, \pi_n.\ \varphi'$ be a formula in $\forall^n$.

Definition 3.10 *Define the 1-reduced formula of φ by*

$$reduce_1(\varphi) := \forall\pi_1, \ldots, \pi_{n-1}. \bigwedge_{i \in 1 \ldots n} \bigwedge_{j \in 1 \ldots i-1} \varphi'[\pi_i \mapsto \pi_j][\pi_n \mapsto \pi_i]$$

The intuition behind this is that in a formula with one path quantifier less, always two paths have to be identical. And therefore, all possible $O(n^2)$ combinations have to be covered with conjunctions.

Theorem 3.11 *It holds that* $\varphi \equiv reduce_1(\varphi)$ *or* $\nexists \phi \in \forall^{n-1}.\ \phi \equiv \varphi$

Proof Suppose there is some $\phi \in \forall^{n-1}$ with $\phi \equiv \varphi$. Now, we will prove that $\phi \equiv reduce_1(\varphi)$.
Let S be an arbitrary set of traces. Let $S' = \{s \mid s \subseteq S, |s| \leq n-1\}$. Because $\phi \in \forall^{n-1}$ and therefore the $n-1$ path variables of ϕ can only talk over $n-1$ paths, $S \models \phi$ is equivalent to $\forall s' \in S'.\ s' \models \phi$, what is by assumption equivalent to $\forall s' \in S'.\ s' \models \varphi$. Now, φ operates on trace sets of only $n-1$ traces which means that at least two path variables of φ have to refer to the same trace. We would now like to prove that $s' \models \varphi \Leftrightarrow s' \models reduce_1(\varphi)$ for any $s' \in S'$.

"⇒": $s' \models \varphi$ implies $s' \models \varphi[\pi_i \mapsto \pi_j]$ for arbitrary i and j because $[\pi_i \mapsto \pi_j]$ is equivalent to restricting π_i and π_j to be the same. Therefore, $s' \models \varphi$ implies $s' \models reduce_1(\varphi)$.

"⇐": Because there are only $n-1$ traces in s', it is always the case that two path variables π_i and π_j with w.l.o.g. $i > j$ refer to the same trace. This behaviour is encoded in the rewriting of $[\pi_i \mapsto \pi_j]$. Writing $\varphi[\pi_i \mapsto \pi_j][\pi_n \mapsto \pi_i]$ is equivalent to enforcing π_i and π_j to be the same. ($[\pi_n \mapsto \pi_i]$ is needed because we do no longer have the variable π_n and need to use the (now) unused variable π_i). If all these rewritings hold on s' at the same time, φ holds on s'.

And the statement $\forall s' \in S'.\ s' \models reduce_1(\varphi)$ is again equivalent to $S \models reduce_1(\varphi)$ because $reduce_1(\varphi)$ only reasons over $n-1$ traces. Because $\phi \equiv reduce_1(\varphi)$ and $\phi \equiv \varphi$ it holds that $\varphi \equiv reduce_1(\varphi)$. □

3.2.3 ∀* in General Undecidable

In the fragment ∀*, we can encode a distributed architecture [10] for LTL synthesis that is undecidable. In particular, we can encode the architecture shown in Figure 3.1. This architecture basically specifies c to depend only on a and analogously d on b. That can be encoded by $D_{\{a\}\mapsto\{c\}}$ and $D_{\{b\}\mapsto\{d\}}$. The LTL synthesis problem for this architecture is already shown to be undecidable [10], i.e., given an LTL formula over $I = \{a, b\}$ and $O = \{c, d\}$ you cannot automatically construct processes p_1 and p_2 that realize the formula.

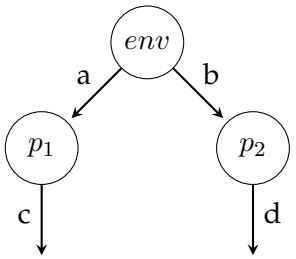

Figure 3.1: An architecture of two processes that specifies process p_1 to produce c from a and p_2 to produce d from b. env is a special environment process that provides the input variables.

Theorem 3.12 *The synthesis problem for ∀* is undecidable.*

Proof by reduction of the LTL synthesis problem in the architecture in Figure 3.1 to the synthesis problem from ∀*.

Let φ be an arbitrary LTL formula. Let $\phi \coloneqq \forall \pi, \pi'.\ D_{\{a\}\mapsto\{c\}} \wedge D_{\{b\}\mapsto\{d\}} \wedge \varphi_\pi$ (where φ_π denotes annotating each proposition in φ with π). It holds ϕ is realizable iff φ is realizable in the architecture in Figure 3.1. This is because $D_{\{a\}\mapsto\{c\}}$ exactly encodes the condition that c only depends on a (d and b analogously). Therefore, the synthesis problem from HyperLTL can not be decided. □

However, in the distributed synthesis from LTL there are architectures that are decidable. These architectures[1] are in some way "linear", i.e., the processes can be ordered such that lower processes always have a subset of the information of upper processes. Coordination Logic [11] embraces exactly this decidable fragment of distributed synthesis. We will characterize a fragment of ∀* that can be transformed to Coordination Logic (CL) and is therefore decidable. But first we have to introduce CL and with that some more theory on trees.

3.2.4 Coordination Logic

Formally, an O labeled (strategy-) tree t over a set of variables I is a mapping $(2^I)^+ \to 2^O$ from an input sequence to an output. For a tree t and a set J disjoint to I we define the J-widening $wide_J(t) : (2^{I \dot\cup J})^+ \to 2^O$ by $in \mapsto t(in|_I)$. Where $in|_I$ denotes intersecting all elements of the sequence in to I. The disjoint union of two O and P labeled trees t and t' over I, $t \dot\cup t' : (2^I)^+ \to 2^{O \dot\cup P}$ is defined by $in \mapsto t(in) \dot\cup t'(in)$. The opposite of widening is narrowing, what is only applicable

[1] ,i.e., architectures without information forks [10]

if the labeling function does not depend on the narrowed variables J. It is defined iff the (at most one) possible t' can be found for $narrow_J(t) \coloneqq t'$ with $t = wide_J(t')$. With these definitions we can define CL[2]. In CL, we have a $\exists c.\varphi$ quantifier. This quantifier introduces a so called coordination variable c. The coordination variables correspond to the input variables. The $\forall$ and $\exists$ quantifier introduce strategies that correspond to the labeling of the tree, i.e., the outputs.

Definition 3.13 (Coordination Logic Syntax)

$$\varphi \coloneqq \exists c.\ \varphi \mid \forall s.\ \varphi \mid \exists s.\ \varphi \mid \text{<LTL formula>}\ \phi$$

with $AP = C \dot\cup S$, $c \in C$ and $s \in S$. In the following, we assume that every formula is well formed, i.e., no coordination variable c and no strategy variable s is introduced twice. We additionally enforce ϕ to contain no free variables[3], i.e., variables that are not introduced by any s or c. As an abbreviation for introducing a set of coordination variables $B = \{b_0 \dots b_n\}$, we use $\exists B$ instead of $\exists b_0 \dots \exists b_n$. We denote with $scope(s)$ all coordination variables that occur left of s in the formula. For the semantics, $\models$ is defined inductively by

Definition 3.14 (Coordination Logic Semantics)

$$\begin{aligned}
t &\models \text{<LTL formula>}\ \phi && \Leftrightarrow t \text{ realizes } \phi \\
t &\models \forall s.\ \varphi && \Leftrightarrow \forall f : 2^{scope(s)} \to 2^{\{s\}}.\ t \dot\cup f \models \varphi \\
t &\models \exists s.\ \varphi && \Leftrightarrow \exists f : 2^{scope(s)} \to 2^{\{s\}}.\ t \dot\cup f \models \varphi \\
t &\models \exists c.\ \varphi && \Leftrightarrow wide_{\{c\}}(t) \models \varphi
\end{aligned}$$

The CL formula φ is valid written $\models \varphi$ iff the empty tree $((x \in \emptyset^+) \mapsto \emptyset) \models \varphi$. In the example $\exists i_{server}\ \exists server\ \exists i_{client}\ \forall client.\ \varphi$, the server sends some information (*server*) based on i_{server} to a client and the client can additionally operate on the information i_{client} to satisfy the formula φ. Intuitively, the coordination variables are the input and the strategy variables are the output as in the LTL synthesis problem. The only restriction is that the strategy variables can only depend on the coordination variables that are already introduced. And that can be encoded in HyperLTL by observational determinism as we will see in the following section.

[2] We actually define a subset where the quantifiers have to be in front of any other operators what corresponds to the restriction for HyperLTL.

[3] That is a restriction of the original definition as now we have formulas forced to be equivalent to either true or false. They would make it much more complicated and we do not need them here.

3.2.5 CL Fragment of ∀*

The high level idea of the characterization of the CL fragment is seeking for variable dependencies of the form $D_{J \mapsto \{p\}}$ with $J \subseteq I$ and $p \in O$ in the formula. If the hyper-part of the formula consists only of such constraints $D_{J_i \mapsto \{p_i\}}$ with the rest being an LTL property, this is the description of an architecture. If furthermore, the $D_{J_i \mapsto \{p_i\}}$ can be ordered such that $J_i \subseteq J_{i+1}$ for all i, the architecture is linear and an equivalent CL formula can be constructed. All in all, there are three main steps to process a given formula φ:

1. First, first we must add general determinism to the formula $\varphi_{det} \coloneqq \varphi \wedge D_{I \mapsto O}$. This does not change realizability as it is always the case in a strategy-tree.

2. Find for each output variable p_i possible sets J_i of variables, p_i depends on. The terms of the form $D_{J_i \mapsto \{p_i\}}$ have to fully describe the hyper-part of the formula. For that, the hyper-part is removed via reduce. We have to find J_i's with $reduce(\varphi) \wedge \bigwedge_{p_i \in O} D_{J_i \mapsto \{p_i\}} \Leftrightarrow \varphi_{det}$. Note that this equivalence is decidable as it is in the ∀* fragment [9]. If we can find such J_i's, φ is an LTL-property together with some dependency descriptions for an architecture and we are in the distributed synthesis setting of LTL. If furthermore, the p_i can be ordered such that $J_i \subseteq J_{i+1}$, we have a linear architecture and we proceed with step three. Note that there are exponentially many possible combination of J_i's.

3. Construct a CL formula φ' that is valid iff φ is realizable. For this, define $J_i' \coloneqq J_i \setminus \bigcup_{k<i} J_k$, $J_{n+1} \coloneqq I \setminus J_n$ and then the CL formula φ':

$$\exists J_1' \; \forall p_1 \; \ldots \; \exists J_n' \; \forall p_n \; \exists J_{n+1}. \; reduce(\varphi)_{LTL}$$

   where $reduce(\varphi)_{LTL}$ denotes removing also the single ∀ and the path annotations at the propositions after application of $reduce()$.

In the following definition, this is subsumed in short words.

Definition 3.15 (CL fragment of ∀*) *A formula φ is in the CL fragment of ∀* iff for all $p_i \in O$ suitable $J_i \subseteq I$ can be found such that*

- $\varphi \Leftrightarrow reduce(\varphi) \wedge \bigwedge_{p_i \in O} D_{J_i \mapsto \{p_i\}}$
- *an ordering of the p_i can be found such that $J_i \subseteq J_{i+1}$ for all i.*

First of all, we observe that $reduce(D_{A \mapsto B}) \equiv true$ for all $A, B \subseteq AP$ because after the application of $reduce$, the formula solely reasons about one single trace and for any proposition $b \in B$ holds $b_\pi \Leftrightarrow b_\pi$.

Note also that each $\forall^1$ formula φ (or φ is reducible to a $\forall^1$ formula) is in the linear fragment because we can set all $J_i = I$ and have additionally $reduce(\varphi) = \varphi$.

As an example, consider $\varphi = \forall\pi, \pi'.\ D_{\{a\}\mapsto\{c\}} \wedge \square(c_\pi \Leftrightarrow d_\pi \wedge b_\pi \Leftrightarrow e_\pi)$ with $I = \{a, b\}$ and $O = \{c, d, e\}$. First, φ has to be "determinized" to $\varphi_{det} = \varphi \wedge D_{I\mapsto O}$. As c and d have to be equivalent, they both have to depend only on a. Possible J's are $\{a\}$ or $\{a, b\}$ for both. But at least one of them has to be $\{a\}$. Otherwise, the J's would not imply the $D_{\{a\}\mapsto\{c\}}$. Note that $D_{\{a\}\mapsto\{d\}}$ suffices to imply the $D_{\{a\}\mapsto\{c\}}$ together with the $\square(c_\pi \Leftrightarrow d_\pi)$. For e we can choose between $\{b\}$ and $\{a, b\}$ where $\{b\}$ would yield an undecidable architecture. All in all we have the equivalence $\varphi_{det} \Leftrightarrow collapse(\varphi) \wedge D_{\{a,b\}\mapsto\{c\}} \wedge D_{\{a\}\mapsto\{d\}} \wedge D_{\{a,b\}\mapsto\{e\}}$. This would yield the CL formula $\exists a\ \forall d\ \exists b\ \forall c, e.\ \square(c \Leftrightarrow d \wedge b \Leftrightarrow e)$ of which the validity can be checked to get the realizability of the initial formula.

Theorem 3.16 *The synthesis from the CL fragment of $\forall^*$ is decidable.*

In the following, we will prove that the previously described decision procedure is correct, i.e., we prove that $\varphi \in \forall^*$ is realizable (w.r.t. the sets I and O) iff the constructed CL formula φ' is valid, assuming φ is in the CL fragment. The procedure clearly always terminates.

Proof Assume $J_i \subseteq I$ are the dependency-sets of the outputs $p_i \in O$ and they are already properly ordered.

"$\Leftarrow$" assume φ' is valid. Have a closer look at the semantic definition of CL. Whenever a strategy variable p_i is introduced, it can only depend on coordination variables in its scope that is J_i. Therefore, the formulas $D_{J_i\mapsto\{p_i\}}$ hold in the tree t that is "constructed" in the definition of CL right before entering the LTL-part of the definition. Obviously, t realizes $reduce(\varphi)$ as it realizes $reduce(\varphi)_{LTL}$. These properties together yield that t realizes $reduce(\varphi) \wedge \bigwedge_{p_i\in O} D_{J_i\mapsto\{p_i\}}$ which is equivalent to φ.

"$\Rightarrow$" assume a tree t realizes φ and therefore $reduce(\varphi) \wedge \bigwedge_{p_i\in O} D_{J_i\mapsto\{p_i\}}$. In particular, t also realizes $reduce(\varphi)_{LTL}$. So we would like to construct t by using the definition of $\exists$ and $\exists$. As in the definition, we start with the empty tree $t' := ((x \in \emptyset^+) \mapsto \emptyset)$ and go from left to right trough the quantifier of φ'.

In the case of an $\exists s$, we choose f to be $narrow_{C\setminus scope(s)}(t)|_s$. Where $|_s$ denotes restricting f to s, i.e., $x \mapsto f(x) \cap \{s\}$. The narrowing is possible because s only depends on its own $J = scope(s)$.

In the case of a $\exists c$, the tree t' is widened as usual what does not change the strategy of the already chosen s. That is because $widen_A(narrow_A(t')) = t'$. Therefore, we can construct the t along the definition of validity of CL and φ' is valid.

Therefore, the constructed CL formula φ' is valid iff φ is realizable. As the validity of CL formulas is decidable [11], the synthesis from the CL fragment is decidable. □

So far, we only analyzed formulas without quantifier alternations. In the next sections, we also look at formulas with quantifier alternations.

3.3 ∀*∃* Fragment

The synthesis from the ∀*∃* fragment turns out to be undecidable. In this section, we will reduce Post's Correspondence Problem (PCP) [17] to the synthesis problem from $\forall^1\exists^1$. As PCP is undecidable, ∀*∃* is undecidable, too.

Theorem 3.17 *The synthesis problem from the ∀*∃* fragment is undecidable.*

In PCP, we are given two lists α and β consisting of finite words from some alphabet Σ. For example, $\alpha = (a, ab, bba)$ and $\beta = (baa, aa, bb)$, where α_i denotes the ith element of the list. One can think of the pairs (α_i, β_i) as domino stones. In our example, we would have the three stones $\frac{a}{baa}$, $\frac{ab}{aa}$ and $\frac{bba}{bb}$. PCP is the problem to find an index sequence $(i_k)_{1\leq k\leq K}$ with $K \geq 1$ and $1 \leq i_k \leq n$ for all k, such that $\alpha_{i_1} \dots \alpha_{i_K} = \beta_{i_1} \dots \beta_{i_K}$. The domino-equivalent is to find a combination of the domino stones such that the word written on the upper half of the stones is the same as the word on the lower half. Assume, you have an infinite amount of every domino stone type. In our example, a possible solution would be $\frac{bba}{bb}\frac{ab}{aa}\frac{bba}{bb}\frac{a}{baa}$ that corresponds to the sequence (3,2,3,1).

To prove the undecidability of the synthesis problem from the ∀*∃* fragment, we give a reduction from an arbitrary PCP instance to a $\forall^1\exists^1$ formula. This proof follows essentially the proof in [9]. There, we have one initial trace that represents a solution to the PCP instance. And we enforce that every trace has a successor,

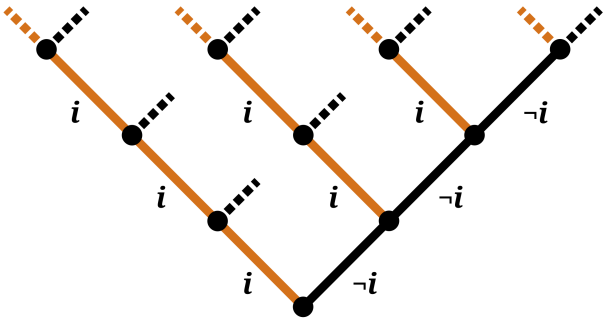

Figure 3.2: The sketch of a strategy-tree. The *relevant* traces are marked orange.

i.e., a trace that is the same as it's predecessor with the first stone removed. In the deterministic synthesis problem, we have only one tree and need to encode this list of successor traces in the tree structure. We do this as showed in Figure 3.2. The $relevant$ traces in which we encode the PCP instance are the orange colored traces in the figure.

Let a PCP instance with $\Sigma = \{x_1, x_2, ..., x_n\}$ and two lists α and β with length m be given. For the synthesis problem, we choose our set of atomic propositions as follows: $AP \coloneqq I \dot{\cup} O$ with $I \coloneqq \{i\}$ and $O \coloneqq (\Sigma \cup \{\dot{x_1}, \dot{x_2}, ..., \dot{x_n}\} \cup \{\dot{\#}, \#\})^2$, where we use the dot symbol to encode that a stone starts at this position of the trace. As abbreviation for $x \vee \dot{x}$ we write $\tilde{x}$ if we do not care if this symbol is an x or $\dot{x}$ and use $*$ as syntactic sugar for an arbitrary symbol out of $\Sigma \cup \{\#\}$. We construct the following HyperLTL formula from the PCP instance:

$$\varphi_{pcp} \coloneqq \forall\pi\exists\pi'.\ \varphi_{sol}(\pi) \wedge (\varphi_{rel}(\pi) \Rightarrow \varphi_{succ}(\pi, \pi') \wedge \varphi_{start}(\bigvee_{i\in[1,m]} (\varphi_{stone_i}(\pi) \wedge \varphi_{shift_i}(\pi, \pi')), \pi))$$

- $\varphi_{sol}(\pi) \coloneqq \square\, i_\pi \Rightarrow (\bigvee_{i=1}^{n} (\tilde{x_i}, \tilde{x_i})_\pi \ \mathsf{U}\ \square(\tilde{\#}, \tilde{\#})_\pi)$ ensures that the trace with globally i as input encodes the solution sequence in the outputs. It also includes that the sequence eventually terminates. This is represented by globally $\tilde{\#}$.
- $\varphi_{rel}(\pi) \coloneqq \neg i\ \mathsf{U}\ \square\, i$ defines the set of $relevant$ traces in our strategy-tree.
- $\varphi_{succ}(\pi, \pi') \coloneqq (\neg i_\pi \wedge \neg i_{\pi'}\ \mathsf{U}\ \square\, i_\pi \wedge \neg i_{\pi'} \wedge \bigcirc \square\, i_{\pi'})$ encodes that a trace π' is a successor of π.
- $\varphi_{stone_i}(\pi)$ encodes that the beginning of trace π represents a stone. An example for the stone $\substack{\text{bba}\\\text{bb}}$ is given below.
- $\varphi_{shift_i}(\pi, \pi')$ encodes that π' is identical to the trace π with the first stone $(stone_i)$ cut off and the rest of the sequence shifted. In contrast to the reduction in [9], we have to add an offset of one to the trace π' as the relevant trace of π' starts one step later. See the example for stone $\substack{\text{bba}\\\text{bb}}$ below.
- $\varphi_{start}(\varphi, \pi) \coloneqq \neg i_\pi\ \mathsf{U}\ \varphi \wedge \square\, i_\pi$ cuts off an irrelevant prefix until the relevant trace starts and asserts φ at the relevant trace.
- We furthermore assume that only one output symbol at once is allowed at any position in the tree. That can be achieved by a disjunction for each pair of the output symbols.

In the following, we give example formulas $\varphi_{stone_i}(\pi)$ and $\varphi_{shift_i}(\pi, \pi')$ for $\substack{\text{bba}\\\text{bb}}$. One can obviously generalize them for arbitrary stones.

$$\varphi_{stone_3}(\pi) \coloneqq ((\dot{b},\dot{b})_\pi \wedge \bigcirc(b,b)_\pi \wedge \bigcirc\bigcirc(a,\dot{*})_\pi \wedge \bigcirc\bigcirc\bigcirc(\dot{*},\tilde{*})_\pi)$$

This encodes that $stone_3$ is at the beginning of trace π followed by either another stone or the end (as $*$ also abbreviates also $\#$).

$$\varphi_{shift_3}(\pi,\pi') \coloneqq \bigwedge_{x\in\Sigma\cup\{\#\}} \Box(\bigcirc\bigcirc\bigcirc(\tilde{x},*)_\pi \Rightarrow \bigcirc(\tilde{x},*)_{\pi'}) \wedge \Box(\bigcirc\bigcirc(*,\tilde{x})_\pi \Rightarrow \bigcirc(*,\tilde{x})_{\pi'})$$

This encodes that the traces π and π' are the same up to the removal of the first stone. Note that the first component of the tuple is shifted by 3 and the second by 2 because the α-part of the stone has length 3 while the β-part has length 2. The following figure shows the shifting of the relevant traces of a strategy tree from the solution $\frac{bba}{bb}\ \frac{ab}{aa}\ \frac{bba}{bb}\ \frac{a}{baa}$ of our initial example.

$(\dot{b},\dot{b})(b,b)(a,\dot{a})(\dot{a},a)(b,\dot{b})(\dot{b},b)(b,\dot{b})(a,a)(\dot{a},a)(\tilde{\#},\tilde{\#})^\omega$ $\frac{\dot{b}ba\dot{a}\dot{b}\dot{b}b\dot{a}\dot{a}}{bb\dot{a}abbbaa}$

$(\dot{a},\dot{a})(b,a)(\dot{b},\dot{b})(b,b)(a,\dot{b})(\dot{a},a)(\tilde{\#},a)(\tilde{\#},\tilde{\#})^\omega$ $\frac{\dot{a}b\dot{b}\dot{b}b\dot{a}\dot{a}}{\dot{a}abbbaa}$

$(\dot{b},\dot{b})(b,b)(a,\dot{b})(\dot{a},a)(\tilde{\#},a)(\tilde{\#},\tilde{\#})^\omega$ $\frac{\dot{b}\dot{b}b\dot{a}\dot{a}}{bbbaa}$

$(\dot{a},\dot{b})(\tilde{\#},a)(\tilde{\#},a)(\tilde{\#},\tilde{\#})^\omega$ $\frac{\dot{a}}{\dot{b}aa}$

$(\tilde{\#},\tilde{\#})^\omega$

Lemma 3.18 *The constructed $\exists\forall$ formula φ' is realizable iff the PCP instance is satisfiable.*

Proof The proof follows the same arguments as the reduction in [9].

"$\Rightarrow$" Assume φ_{pcp} is realizable. Then the trace with $\Box\, i$ represents the solution-sequence to the PCP instance. To extract the sequence of stones, we have to analyze which stones are encoded at the beginning of the relevant traces. The stone at the first relevant trace, i.e., the solution-trace is the first stone. Then we iteratively look at the successor traces (φ_{succ}). Their beginnings represent the second, third, ... stone in the sequence.

"$\Leftarrow$" Assume, the PCP instance has the solution $(i_1, i_2, \ldots, i_k)$ with the resulting word $w = \alpha_{i_1} \ldots \alpha_{i_k} = \beta_{i_1} \ldots \beta_{i_k}$. Then there is a strategy-tree t with w written at it's solution-trace π with $\Box\, i$. The beginning of a stone is always marked with the dot over a symbol. The successor trace of π encodes the shifted word with the stone i_1 removed as described in φ_{shift_i}. One can see, that t can obviously fulfill the constraints in the formula and, therefore, t realizes φ_{pcp}. $\square$

Proof (of Theorem 3.17) As PCP is undecidable and we gave a reduction from an arbitrary PCP instance to a $\forall^1\exists^1$ formula, $\forall^*\exists^*$ is undecidable. $\square$

Note that this also yields that formulas with at least one alternation from $\forall$ to $\exists$ are undecidable.

Chapter 4

Related Work

4.1 Model Checking

Model checking [5] is a technique that verifies the correctness of a given system with respect to a given formula. The system is usually given as a transition system. Transition systems consist of labeled states and transitions between them. The possible executions of a transition system are defined by the sequences of states that can be walked along via the transitions from an initial state. The produced traces are the label sequences of the states of the executions. See Figure 4.1 for an example.

LTL model checking is done with an automata based algorithm. There, we construct a nondeterministic Büchi automaton from the negated LTL formula. After that, we determinize the automaton and compute the cross product with the system. The result of the cross product is an automaton that accepts exactly the traces that are produced by the system and are at the same time invalid according to the formula because we negated the formula in the beginning. Therefore, the automaton is empty iff the system is correct, i.e., all system traces fulfill the specification.

HyperCTL* [7] is a logic that extends HyperLTL. In contrast to HyperLTL where quantifiers are only located in the quantifier prefix, it allows the use of quantifiers inside the formula. This enables HyperCTL* to reason about branching properties of a system as illustrated in Figure 4.1.

Model checking of HyperCTL* can be done as well as HyperLTL model checking and is fully decidable [12]. The idea is to construct bottom up a series of automata. The lowest automaton reasons about n-tuples of traces if n is the amount of quantifiers above it. The automaton for the whole formula is inductively constructed bottom up. For temporal and propositional operators, the automaton is constructed as in LTL. For existential quantifiers, the resulting automaton is a cross product of the automaton from the direct subformula and the transition system. This new automaton reasons only about n-1-tuples of traces. However, because universal quantifiers are handled via negated existential quantifiers, the automata has to be negated for each quantifier alternation. This leads to an exponential blow-up of the state amount of the automaton. Therefore, the model checking problem

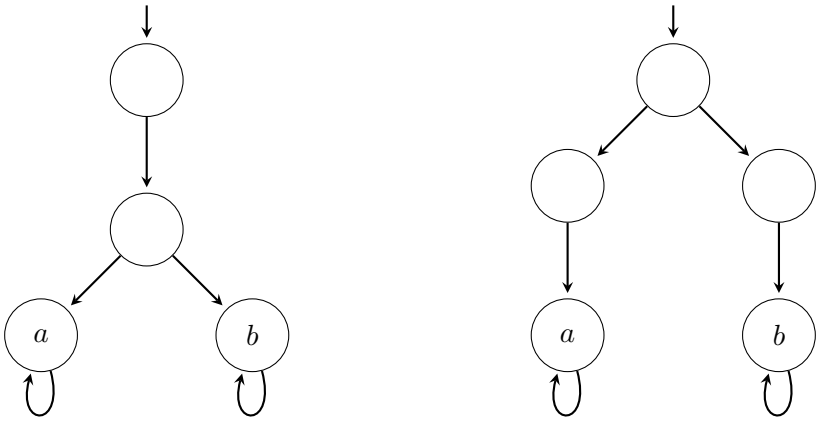

Figure 4.1: These two transition systems cannot be distinguished by an LTL or HyperLTL formula as they produce the same traces $\{\emptyset\emptyset a^\omega, \emptyset\emptyset b^\omega\}$. However, they can be distinguished in CTL* and HyperCTL* by $\forall\pi.\bigcirc\exists\pi'.\bigcirc a_{\pi'}$ which is satisfied by the left system but not by the right system.

of HyperCTL* is decidable in non-elementary time and space. As a corollary, we get the result that model checking of the alternation free fragment, the procedure is in PSPACE [12].

4.2 LTL Synthesis

LTL synthesis [16] can also be done by using automata. We have already seen that it is possible to construct an automaton that accepts all words which are in the specified language of an LTL specification. A realization of an LTL specification would be a strategy-tree whose traces are all accepted by the automaton. This leads to the so called tree automata. A tree automaton is an automaton that runs on trees instead of traces. From an arbitrary automaton on traces, a tree automaton can be constructed that accepts a tree iff all its traces are accepted from the initial automaton. The emptiness of the tree automaton that is equivalent to the non-existence of a solution to the synthesis problem can be checked by reduction to an infinite game [16]. Therefore, the synthesis problem from LTL is decidable.

4.3 Distributed Synthesis from LTL

Distributed synthesis from LTL [10] is the problem of finding multiple implementations of processes in an architecture that together fulfill an LTL formula. As already said, there are some architectures for which synthesis is undecidable. These architectures are architectures that contain a so called information fork [10]

between two processes. Intuitively, that is the case if two processes (directly or indirectly) get different information from the environment. For example this is the case in Figure 3.1 where process p_1 receives a but not b and vice versa for process p_2.

If an architecture contains no information fork, we can order the processes according to their level of information, i.e., the environment/input variables that they receive (direct or indirect). To decide the distributed synthesis problem from LTL in such an architecture one has to perform two steps. First, one transforms the architecture into a linear architecture. In such a linear architecture, processes with the same level of information are collapsed to one single processes and the resulting processes are ordered according to their level of information. At the end, a process with a lower level of information has always a subset of the information of a higher level process. In the second step, the processes are handled in order of their level of information. For the first process, i.e., the process with the most information a tree automaton is constructed. For each of the following processes, the automaton is narrowed to fit to the informedness of the lower process. At the end we have to do an emptiness check as in the LTL synthesis procedure.

4.4 Coordination Logic

With these linear architectures in mind, we can have a look at Coordination Logic [11]. Coordination Logic is like synthesis of LTL but the strategy variables are limited to depend only on the coordination variables that are already introduced. Therefore, the strategy variables are also ordered by their level of information. Outer strategy variables have always a subset of the information of inner strategy variables. Therefore it is clear that each Coordination Logic formula can simply be encoded in an distributed synthesis problem without information fork. The strategy variables correspond to the processes and the coordination variables correspond to the environment variables.

4.5 Bounded Synthesis from LTL

When synthesizing LTL specifications, the resulting system has no limitations in terms of size, i.e., it is not guaranteed that the system produced by the synthesis procedure is minimal. However, one usually seeks for small implementations to minimize costs of, e.g., a hardware circuit. Bounded synthesis [19] is an approach to solve this problem. In bounded synthesis, we seek for implementations that are not larger than a given bound b. This search can be repetitively executed to find the lowest possible bound which yields the minimal solution to the synthesis problem. Another aspect of bounded synthesis is the decidability. As we talk only about systems with a limited number of states, the synthesis problem is decidable even for distributed architectures with information forks.

In bounded synthesis from LTL, first of all, a universal co-Büchi tree automaton is constructed. Co-Büchi means here that there are some rejecting states in the automaton that are forbidden to be visited infinitely often in a run. Then, the cross product with an transition system that we gather later is constructed. This yields again an automaton. In this stage, so called annotation functions are introduced. An annotation function λ annotates each state of an automaton with a natural number. Additionally, for each transition $q \mapsto q'$ of the automaton, $\lambda(q) \leq \lambda(q')$ it has to hold. If q' is a rejecting state, it has $\lambda(q) < \lambda(q')$ has to hold. This ensures that $\lambda(q)$ indicates the maximal amount of rejecting states can occur on a path from the initial state to q. If there is such an annotation function for an automaton, one can clearly see that there is no loop containing a rejecting state because otherwise the annotations in this loop have to be always increasing. The absence of such a loop yields that every tree is accepted by the automaton. Therefore, there exists such an annotation function if and only if the transition system from the cross product realizes the formula from which the automaton is constructed.

However, we do not have this ominous transition system to construct the cross product with te automaton. Here, a second important technique comes into play. We will encode the problem as a SAT instance. The constraints of the SAT instance ensure that some uninterpreted function symbols represent a transition system. And some other constraints ensure that there exists a valid annotation function on the cross product of the transition system and the automaton. These two constraints yield that the boolean formula is satisfiable iff there exists a transition system that realizes the given formula. This transition system can easily be extracted from a solution to the satisfiability problem.

Chapter 5

Conclusion

Finally, we can say that we analyzed the ∀* fragment and the ∃* fragment exhaustively. The synthesis for the ∃* fragment is fully decidable and for the ∀* fragment, it is partially decidable. That is a new contribution to the research on HyperLTL. We can now synthesize HyperLTL properties in ∃* and the linear ∀* fragment. In the following, we shortly revisit the example from the beginning. There, we had an arbiter that had n request inputs and n grant outputs. The goal was to schedule grants in a way that every request is eventually answered by a grant: $\square(request_i \Rightarrow \Diamond\, grant_i)$ globally for all i and mutual exclusion: $\neg(request_i \wedge request_j)$ globally for all $i \neq j$. To enforce observational determinism, we could very restrictively say that every grant solely depends on its request. Sadly, this is not contained in the CL fragment that we can reduce to the decidable Coordination Logic because the dependencies cannot be ordered, i.e., these dependencies build an architecture that is not linear and therefore not decidable. Note that despite this undecidability, there exists an implementation that realizes these constraints by simply enabling one grant after another (independently from all inputs), i.e., undecidability does not imply unrealizability. Here we see the limits of decidability very clear.

However, we can recover this example by requiring observational determinism only in a hierarchical manner. We can for example say "process i has more rights than process j iff $i > j$". In this case, we would like to have the property that process j cannot read information about i's request signal because i is higher ranked than j. That is again a property that could be synthesized with the methods of this thesis as it is in the CL fragment.

Other achievements of this thesis are the reductions to reduce the quantifier amount in the ∀* fragment. Imagine the following scenario: Someone would like to write a specification and it contains three ∀ quantifiers. When it comes to model checking, the computation time exceeds all limits because in model checking, each quantifier leads to a costly self composition of the constructed automata. After some time, the computation is aborted by the impatient user. With the reduction from $\forall^3$ to $\forall^2$ the model checker could warn the user that the property he inserted can also be written with only two quantifiers. After a short time of pondering, the user would write a smaller formula or would use the suggestion that the reduc-

tion gave him. In this case, the model checker would finish after a short time of calculation.

It is worth mentioning that it was unusual to search for nondeterministic solutions as we usually only search for deterministic systems to implement. However, nondeterministic solutions are interesting because we can express nondeterminism in the logic. Wy should we then forbid it as a valid solution to give multiple strategy trees? It is also more natural to allow nondeterministic solutions instead of a deterministic "restriction". This shows up in the $\exists^*$ section. The nondeterministic synthesis is directly equivalent to satisfiability while we have to add an extension to the formula in the deterministic case.

5.1 Future Work

As a future work, I would propose to have a look at the $\exists^*\forall^*$ fragment. It is already proven [3] that the realizability of the $\exists^*\forall^1$ fragment is decidable while it is undecidable for the $\exists^*\forall^{>1}$ fragment. Maybe there is an even bigger decidable fragment inside $\exists^*\forall^*$ analogous to the CL fragment of $\forall^*$ (that fully contains the $\forall^1$ fragment).

Bounded synthesis as mentioned in related work can be extended to HyperLTL. A first approach has already been done [3], although it covers only the $\forall^*$ fragment. It could be extended to the full logic of HyperLTL.

CTL* [4, 8] is an extension to LTL that is able to refer to branching properties. There also exists a lifting to HyperCTL* that we discussed in the related work. A resulting future work would be to have a look at HyperCTL* synthesis. One could also invest in finding some fragments for the satisfiability of HyperCTL*, that is undecidable in general as it embeds HyperLTL.

Bibliography

[1] Hartbleed Vulnerability. http://heartbleed.com/, 2014. [Online; accessed 28-January-2018].

[2] Syntcomp. http://www.syntcomp.org/, 2018. [Online; accessed 5-February-2018].

[3] Philip Lukert Marvin Stenger Bernd Finkbeiner, Christopher Hahn and Leander Tentrup. Synthesizing reactive systems from hyperproperties. In *[under review for publication]*, February 2018.

[4] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. ISBN 978-0-262-03270-4. URL http://books.google.de/books?id=Nmc4wEaLXFEC.

[6] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=1891823.1891830.

[7] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *POST*, pages 265–284, 2014.

[8] E Allen Emerson and Joseph Y Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[9] Bernd Finkbeiner and Christopher Hahn. Deciding Hyperproperties. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-017-0. doi: http://dx.doi.org/10.4230/LIPIcs.CONCUR.2016.13. URL http://drops.dagstuhl.de/opus/volltexte/2016/6170.

[10] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *IEEE Symposium on Logic in Computer Science*, pages 321–330, June 2005.

[11] Bernd Finkbeiner and Sven Schewe. Coordination logic. In *CSL*, pages 305–319, 2010.

[12] Bernd Finkbeiner, Markus N. Rabe, and César 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*, pages 30–48, 2015. doi: 10.1007/978-3-319-21690-4_3. URL `https://doi.org/10.1007/978-3-319-21690-4_3`.

[13] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[14] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[15] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977. doi: 10.1109/SFCS.1977.32.

[16] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75293. URL `http://doi.acm.org/10.1145/75277.75293`.

[17] Emil L Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[18] A. Roscoe. Csp and determinism in security modelling. In *Proceedings 1995 IEEE Symposium on Security and Privacy(SP)*, volume 00, page 0114, 05 1995. doi: 10.1109/SECPRI.1995.398927. URL `doi.ieeecomputersociety.org/10.1109/SECPRI.1995.398927`.

[19] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *Proc. ATVA*, pages 474–488. Springer-Verlag, 2007.

[20] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *In Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, 2003.