Saarland University
Faculty of Mathematics and Computer Science

Bachelor's Thesis

Temporal Stream Logic for Hyperproperties

Author
Julia Tillman

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Advisors
Jana Hofmann, M.Sc.
Norine Coenen, B.Sc.

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr.-Ing. Holger Hermanns

Submitted: November 3rd, 2020

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig 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, November 3rd, 2020

Abstract

Temporal logics are commonly used in verification and synthesis to formally specify the desired behavior of a system. While the most popular temporal logics, LTL and CTL*, can express properties on systems with finite data, they cannot specify properties on systems with infinite data. This is because they discretize data and encode it using atomic propositions. Temporal Stream Logic, TSL, on the other hand, abstracts from concrete data by using predicates and functions instead of atomic propositions. Therefore, it can express properties on systems with infinite data. Recent years have shown that temporal logics sometimes do not suffice to specify the desired behavior of a system because many important information flow properties in software are hyperproperties. Hyperproperties specify relationships between multiple execution traces and are commonly specified using temporal hyperlogics as standard temporal logics like LTL or CTL* cannot reason about several traces. Popular temporal hyperlogics are HyperLTL and HyperCTL*, which extend LTL and CTL* with explicit quantification over traces. As they are based on LTL and CTL*, they cannot specify hyperproperties on systems with infinite data. Therefore, information flow properties in software with infinite data cannot be expressed using HyperLTL or HyperCTL*. In this thesis, we introduce the hyperlogic HyperTSL which extends TSL with explicit quantification over traces. Since it is based on TSL, HyperTSL can express hyperproperties on software with infinite data. We identify a syntactical fragment of HyperTSL, HyperTSL$^-$, and develop a sound bounded synthesis approach for the universal fragment of HyperTSL$^-$. Furthermore, we present an application of HyperTSL in the field of software doping, where we use the hyperlogic to spot deliberate manipulation of software by the provider to perform against the best interest of the user.

Acknowledgements

First, I want to thank Professor Finkbeiner for providing me with this topic and for the opportunity to write my thesis at his chair. Furthermore, I want to thank my advisors Jana Hofmann and Norine Coenen for their time, advice and amazing support during the course of this thesis. In addition, I want to thank Professor Hermanns for reviewing this thesis and for a great time at his chair during the past two semesters. Special thanks also go to Sebastian Biewer for discussing details of his research in software doping with me.

Lastly, I want to thank my family and friends for their ongoing support. Special thanks go to Niklas and Gregory for their support and to Lena and Jannis for proofreading this thesis.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Related Work 5
1.2 Structure 6

2 Preliminaries **7**
2.1 LTL 7
2.1.1 Syntax 8
2.1.2 Semantics 8
2.2 HyperLTL 9
2.2.1 Syntax 9
2.2.2 Semantics 10
2.3 HyperLTL Synthesis 11
2.4 TSL 13
2.4.1 Syntax 16
2.4.2 Semantics 16

3 HyperTSL **18**
3.1 Traces 20
3.2 Battery Example 25
3.3 HyperTSL$^-$ 28

4 Use Case: Diesel Scandal **32**
4.1 The NEDC 33
4.2 Specifying the System 34
4.3 Specifying Hyperproperties 38

5 Synthesis **43**

6 Conclusion **57**
6.1 Future Work 58

Bibliography **59**

Chapter 1

Introduction

Software surrounds us almost everywhere in our lives. Very often, it fulfills critical functions, making it crucial that the software behaves as intended. In order to make software realize the intended behavior, we use temporal logics that specify properties on software traces. Linear Temporal Logic (LTL) [17] and Computational Tree Logic (CTL*) [6] are two popular temporal logics that succeed in specifying trace properties on software with finite data. Often, software needs to interact with its environment, for instance, when it must rely on sensor data. As sensors constantly provide new inputs, large amounts of data result. If software gets inputs from several sensors, the overall amount of data grows even larger. Software thus does not only face the challenge of fulfilling the specified properties but also must manage lots of data in the process. To specify behavior of software with infinite data, we again need temporal logics. The temporal logics LTL and CTL* can, however, not express properties on software with infinite data, as they discretize data and encode it as atomic propositions. Temporal Stream Logic, TSL, is a recently developed logic that allows expressing properties on software with infinite data [10]. In contrast to LTL and CTL*, it abstracts from concrete data and uses predicates and functions instead of atomic propositions to represent data.

In addition to fulfilling a specification expressed in a temporal logic, we also want software to fulfill certain security requirements, like not leaking data that is supposed to remain secret. Many important properties of software can be expressed using trace properties, which describe how a system should behave under specific circumstances over time. Another type of properties on software that cannot be specified using trace properties are hyperproperties. Hyperproperties are sets of sets of traces that relate several executions and describe how they behave in comparison to each other. Many important information flow and security properties on software are hyperproperties. Observational determinism, for instance, is a hyperproperty stating that for all pairs of traces, it must hold that if the observable inputs are the same then the observable outputs are also the same [8]. This ensures that executions with the same observable inputs but different secret inputs do not cause different observable outputs and through that allow inferring secret information.

While temporal logics like LTL and CTL* are very common specification languages for describing standard system properties, they are not sufficiently expressive in some cases. Both LTL and CTL* cannot express properties on software with infinite data, as they represent data through atomic propositions. When it comes to expressing hyperproperties, the two logics also fall short as they can only reason about one trace at a time.
When describing software behavior, we likely need to reason about infinite data and also specify behavior that relates several executions. While TSL can reason about infinite data in software, it cannot specify hyperproperties. Therefore, in this thesis, we introduce HyperTSL, which is a hyperlogic extending TSL to express hyperproperties. HyperTSL is inspired by the popular hyperlogic HyperLTL.
HyperLTL uses explicit quantification over traces, where traces are denoted using trace variables [2]. Atomic propositions occurring in HyperLTL formulas are indexed with these trace variables, indicating from which trace they stem. Apart from a quantifier prefix and indexed atomic propositions, HyperLTL formulas follow the same syntactical structure as LTL formulas. For instance, a HyperLTL formula expressing observational determinism [8] looks like this:

$$\forall\pi\forall\pi'.\,\square\,(i_\pi \leftrightarrow i_{\pi'}) \rightarrow \square\,(o_\pi \leftrightarrow o_{\pi'}) \tag{1.1}$$

The indexed atomic proposition i_π denotes the observable inputs of trace π and $i_{\pi'}$ denotes the observable inputs of trace π' and o_π and $o_{\pi'}$ denote the corresponding observable outputs of traces π and π', respectively. The formula states that for two traces, if the observable inputs are always the same, then the observable outputs must also always be the same. This hyperproperty enforces that secret inputs to a system do not influence the observable outputs.
While HyperTSL is largely inspired by HyperLTL, another important component of HyperTSL is TSL. Due to the fact that HyperTSL extends TSL, it inherits all the advantages of the temporal stream logic: TSL abstracts from concrete data and can thus express properties of software involving infinite data. It uses *predicates* that check if certain conditions hold on an argument to guide the control flow. TSL uses a finite number of cells to store values. *Updates*, which are of the form $[\![\texttt{cell} \leftarrowtail \texttt{value}]\!]$, are used to store a value in a cell. In one time step, a cell can always only be assigned one value. An update $[\![\texttt{cell} \leftarrowtail \texttt{value}]\!]$ in a formula evaluates to true if it was indeed the case that `value` was assigned to `cell` in that time step and nothing else. In the respective next time step, `value` can be read from the cell and used for further computations. To illustrate the use of the logic, we state two exemplary TSL formulas that describe properties of a system that consumes energy from a battery if the input is above a given threshold and that turns on an alarm when the battery value is below a certain value.

$$\square\,(\texttt{greater}(\texttt{needsEnergy}, \texttt{t}) \leftrightarrow [\![\texttt{bat} \leftarrowtail \texttt{dec bat}]\!]) \tag{1.2}$$

$$\square\,(\texttt{below15}\ (\texttt{bat}) \leftrightarrow [\![\texttt{critical} \leftarrowtail \texttt{true}]\!]) \tag{1.3}$$

The first formula states that whenever the value provided by the input stream `needsEnergy` is greater than some predefined threshold `t`, then and only then the cell `bat` is decremented by one. This means that energy was consumed. The second formula states that whenever the battery value is `below15`, so below 15 %, then and only then the cell `critical` is updated with `true` to indicate that the battery value is critically low. Using cells, TSL can store data from a specific point in time and thus make it accessible even several time steps later. Through predicates and functions, TSL abstracts from concrete data and can therefore reason about data from infinite domains. Hence, TSL is more expressive than LTL or CTL*.

With HyperTSL, we have a logic that combines the advantages of TSL with the expressivity of a hyperlogic. Predicates and updates in a TSL formula are each composed of different components which makes it not obvious how a hyperlogic for TSL looks like. Especially for indexing predicate terms, there are several design possibilities which is why we introduce HyperTSL and a fragment of it, HyperTSL$^-$. HyperTSL allows predicates over arguments from different traces, and HyperTSL$^-$ only allows predicates to take arguments from the same trace. Similar to HyperLTL, a HyperTSL formula starts with a quantifier sequence and uses trace variables as indices to denote what trace a component stems from. In LTL, there are only atomic propositions, so the straightforward approach is to index the atomic propositions. In TSL however, predicates and updates consist of several components. Therefore, there are different possibilities for indexing. For instance, we can treat a predicate and its arguments as one atomic component and index it with a trace variable. We can do the same for updates. This approach results in the logic HyperTSL$^-$, which seems to have the most in common with HyperLTL when it comes to atomic components.
Another approach is to treat predicates and their arguments separately. The result is the hyperlogic HyperTSL, where each argument of a predicate is indexed individually. A similar approach for updates, where the cell and the function term are indexed individually, however, cannot be taken. The resulting logic would allow expressing hyperproperties where a function term could be assigned to a cell from a different trace. This would describe behavior that is not allowed by TSL itself. Therefore, the only option is to see updates as atomic components and index them accordingly. We specify a hyperproperty using HyperTSL on the system we specified by the Formulas 1.2 and 1.3:

$$\begin{aligned} &\forall\pi\forall\pi'.\,\square\ \texttt{equal}(\texttt{needsEnergy}_\pi, \texttt{needsEnergy}_{\pi'}) \\ &\quad\rightarrow \square\ (\llbracket \texttt{critical} \leftarrowtail \texttt{true} \rrbracket_\pi \leftrightarrow \llbracket \texttt{critical} \leftarrowtail \texttt{true} \rrbracket_{\pi'}) \end{aligned} \tag{1.4}$$

This hyperproperty expresses that for a pair of traces where the inputs provided by `needsEnergy` are always the same, the `critical` cell is updated with `true` in exactly the same time steps on both traces. To compare the values of `needsEnergy` from both traces we use the uninterpreted predicate `equal` which, for this example, we treat as the equality predicate.

Using HyperTSL$^-$, we can express a similar hyperproperty on the system, for instance:

$$\forall\pi\forall\pi'.\,\square\,([\texttt{greater}(\texttt{needsEnergy},\texttt{t})]_\pi \leftrightarrow [\texttt{greater}(\texttt{needsEnergy},\texttt{t})]_{\pi'}) \quad (1.5)$$
$$\rightarrow \square\,[\![\texttt{critical} \leftrightarrow \texttt{true}]\!]_\pi \leftrightarrow [\![\texttt{critical} \leftrightarrow \texttt{true}]\!]_{\pi'}$$

In contrast to the previous hyperproperty, we cannot check for equality of the input streams as this would require evaluating a predicate over arguments from different traces. Instead, we can use the `greater` predicate to check whether the inputs are greater than the threshold and, using an equivalence, we can state that if the inputs on both traces always evaluate to the same value under the predicate `greater` then the `critical` cells should also behave in the same way. Note that we only use predicates that take arguments from one trace here, whereas in the previous formula (1.4), we used a predicate that took arguments from two different traces. We suppose that identifying HyperTSL$^-$ as a fragment of HyperTSL can be especially helpful when considering problems such as synthesis. For instance, approximating HyperTSL$^-$ with HyperLTL creates fewer challenges than approximating HyperTSL with HyperLTL, as the basic components of a HyperTSL$^-$ formula are more similar to the components of a HyperLTL formula because all information contained in one component comes from one trace. For HyperTSL, however, the information contained in a basic component is potentially composed from multiple traces. We further explore this subject in Chapter 5.

An example where HyperTSL and HyperTSL$^-$ can express many useful hyperproperties in addition to what is already specified in HyperLTL is found in software doping [4]. This is because due to predicates and updates, HyperTSL and HyperTSL$^-$ can express hyperproperties on software with infinite data and are thus more expressive than HyperLTL. Software doping describes scenarios where software is deliberately designed to perform worse or against the best interest of the user, in order to be more profitable for the company providing the software. Such behavior can only be discovered when considering several executions and comparing the behavior. Examples of software doping can be found in technical devices such as printers, phones, or laptops that do not work anymore if attached to supplies of a different brand. This binds the user to the company or makes him buy replacements earlier than needed, thus benefitting the company. Another major example of software doping is the Diesel exhaust emission scandal, where software was manipulated to produce low emissions when attached to testing stations but emit high emissions when actually in use. In Section 4, we explore the Diesel emission example further and use HyperTSL and HyperTSL$^-$ to express hyperproperties that spot software doping prior to use.

TSL was initially designed for the synthesis of reactive systems [10]. Synthesis is the process of constructing a system from a specification where the resulting system is guaranteed to fulfill the specification by design. Synthesis is where we can see another major advantage TSL has over LTL: It can express large LTL specifications more concisely and allows for synthesizing specifications that could not be synthesized

using LTL. However, higher expressivity of TSL in synthesis comes at the cost of decidability. While the resulting system is guaranteed to fulfill the specification, there is no guarantee of whether it fulfills certain hyperproperties as well. While a bounded synthesis approach for HyperLTL already exists, there is not yet an approach for synthesizing a TSL specification with hyperproperties. However, this possibility is especially attractive since some LTL specifications could not be synthesized while the same specification expressed in TSL could. In order to specify hyperproperties on exactly these systems, a hyperlogic for TSL is needed, since HyperLTL does not fit the underlying specification language. In this thesis, we present a bounded synthesis approach for formulas in the universal ($\forall^*$) fragment of HyperTSL$^-$. The universal fragment of HyperTSL$^-$ contains all HyperTSL$^-$ formulas, that contain only universal quantifiers. We approximate the universal HyperTSL$^-$ formula through a universal HyperLTL formula and try to construct a sound strategy based on existing bounded synthesis approaches for HyperLTL. We can thus, not only use HyperTSL$^-$ to express system specifications, but we can also directly construct systems that fulfill the desired hyperproperties.

1.1 Related Work

HyperLTL, as introduced by Clarkson et al. [2], is a popular temporal hyperlogic. Many important security and information flow properties, like noninterference or observational determinism, can be specified using HyperLTL. HyperLTL synthesis was explored by Finkbeiner et al. in [11]. While HyperLTL synthesis is in general undecidable, some decidable fragments were identified and a bounded synthesis tool, BoSyHyper, was implemented. Furthermore, model checking algorithms for HyperLTL were developed by Finkbeiner et al. [9].

TSL, temporal stream logic, was introduced by Finkbeiner et al. [10]. The syntax, semantics, and term notation we use for HyperTSL are based on the syntax, semantics, and term notation of TSL. TSL was mainly developed for the synthesis of reactive systems. Several specifications have been successfully synthesized, which was not possible when using the standard temporal logic LTL. The synthesis approach described by Finkbeiner et al. approximates the TSL formula to an LTL formula and tries to construct a sound realizing strategy using standard LTL synthesis tools as, for instance, the bounded synthesis tool BoSy. In this thesis, we describe a bounded synthesis approach for universal HyperTSL$^-$ inspired by the bounded synthesis approach for TSL used by Finkbeiner et al. [10]. While for TSL synthesis Finkbeiner et al. used an approximation to LTL, for HyperTSL$^-$ synthesis we use an approximation to HyperLTL. Synthesis for hyperproperties and especially for HyperLTL is described by Finkbeiner et al. in [11]. We define HyperTSL realizability based on the notion of HyperLTL realizability [11] and TSL realizability [10]. We adapt both the approximation and the spuriousness checks that were used for TSL synthesis [10] to work for hyperproperties. The spuriousness check looks for inconsistent predicate evaluations of two equal

predicates by the strategy along a branch in the strategy tree. The hyper-spuriousness check needs to do the same but this time for multiple branches. Our approach also exploits that there is an already existing bounded synthesis tool for hyperproperties, BoSyHyper [11]. As we describe a bounded synthesis approach, it is not complete. The approximation from universal HyperTSL$^-$ to HyperLTL however is sound as we prove in Proof 5.2.

1.2 Structure

This thesis is structured as follows: In Chapter 2, we lay the foundations regarding the temporal logics we use in this thesis as well as HyperLTL synthesis. We extend TSL to express hyperproperties and formalize the new hyperlogic HyperTSL and its fragment HyperTSL$^-$ in Chapter 3. In Chapter 4, we present a use case for HyperTSL and express hyperproperties in the context of software doping. We describe a bounded synthesis approach for universal HyperTSL$^-$ and prove that the approximation we use is sound in Chapter 5. We conclude in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we lay the foundations to understand the construction of the hyperlogic HyperTSL. We first have a look at LTL and HyperLTL to see how a hyperlogic is constructed from a logic. Then, we shortly discuss HyperLTL synthesis and introduce TSL.

2.1 LTL

Linear temporal logic, LTL, is a popular logic used in formal verification. It was first introduced by Amir Pnueli in 1977 [17]. We have a look at LTL as described by Finkbeiner et al. [8]. We assume time to be discrete and represent it through the set $\mathbb{N}$ of natural numbers. Let AP be a finite set of atomic propositions. A trace $tr \in (2^{AP})^\omega$ fixes for each point in time which set of atomic propositions holds. The set of all traces over AP is denoted by $(2^{AP})^\omega$.

Example 2.1
As an example, we fix $AP = \{a, b\}$. Then a trace tr over AP can be expressed by the sequence $tr(0)tr(1)tr(2)\ldots$. Assuming that in the first time step no atomic proposition holds, then in the second one both hold and in the third one only a holds, it can be rewritten to $\emptyset\{a, b\}\{a\}\ldots$. This trace is illustrated in Figure 2.1.

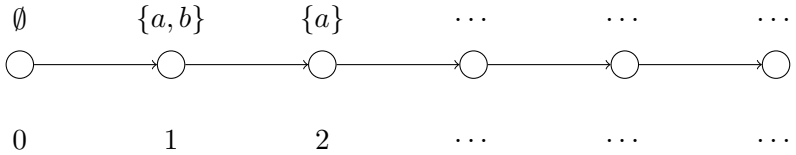

Figure 2.1: An LTL trace

Definition 2.1 (Trace Projection)
We define the projection of a trace tr over the set AP onto a set S with $S \subseteq AP$ to be the sequence $\nu \in (2^S)^\omega$ such that $\forall t \in \mathbb{N}.\ \nu(t) = tr(t) \cap S$.

2.1.1 Syntax

Definition 2.2 (Syntax)
An LTL formula is constructed using the following grammar:

$$\varphi \coloneqq \ a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \ \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi$$

LTL formulas contain atomic propositions, represented by the atomic proposition $a \in AP$ in the grammar, that are either *true* or *false*. Apart from the standard boolean operators like *negation* ($\neg$) and *conjunction* ($\wedge$), we also have the temporal operators *next* ($\bigcirc$) and *until* ($\mathcal{U}$). The temporal *next* operator $\bigcirc\varphi$ expresses that after one time step φ must hold, and the *until* operator $\varphi \,\mathcal{U}\, \psi$ expresses that at some point in time ψ must hold but up until this point φ must hold. Apart from the standard boolean and temporal operators, we also have the following derived boolean and temporal operators:

- $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$, states that φ *or* ψ must hold
- $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$, states that φ *implies* ψ
- $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$, states that φ is *equivalent* to ψ, that is φ holds *if and only if* ψ holds
- $\varphi \oplus \psi \equiv (\neg\varphi \wedge \psi) \vee (\varphi \wedge \neg\psi)$, states that *either* φ *or* ψ holds
- $\varphi \,\mathcal{R}\, \psi \equiv \neg(\neg\psi \,\mathcal{U}\, \neg\varphi)$, states that either ψ always holds or that ψ holds until at some point $\psi \wedge \varphi$ hold, i.e., φ *releases* ψ
- $\Diamond\varphi \equiv \mathit{true} \,\mathcal{U}\, \varphi$, states that *eventually* φ will hold
- $\Box\varphi \equiv \mathit{false} \ \mathcal{R}\, \varphi$, states that φ *always* holds
- $\varphi \,\mathcal{W}\, \psi \equiv (\Box\varphi) \vee (\varphi \,\mathcal{U}\, \psi)$, states that either φ always holds or that eventually ψ holds but up to this point φ holds

2.1.2 Semantics

Let $tr \in (2^{AP})^\omega$ be a trace. The satisfaction of an LTL formula φ is evaluated with respect to a trace tr and a point in time $t \in \mathbb{N}$ via the following semantics:

Definition 2.3 (Semantics)

$$\begin{aligned}
&tr, t \models a && \Leftrightarrow a \in tr(t) \\
&tr, t \models \neg\varphi && \Leftrightarrow tr, t \not\models \varphi \\
&tr, t \models \varphi_1 \wedge \varphi_2 && \Leftrightarrow tr, t \models \varphi_1 \ \wedge \ tr, t \models \varphi_2 \\
&tr, t \models \bigcirc\varphi && \Leftrightarrow tr, t+1 \models \varphi \\
&tr, t \models \varphi_1 \,\mathcal{U}\, \varphi_2 && \Leftrightarrow \exists i \geq t.\ tr, i \models \varphi_2 \ \wedge \ \forall t \leq j < i.\ tr, j \models \varphi_1
\end{aligned}$$

An atomic proposition a holds w.r.t. a trace tr and a point in time t iff it is contained in the set of atomic propositions that hold on trace tr at time point t. In order to check whether a formula $\neg\varphi$ is satisfied by a trace and a point in time, we check whether the formula φ is not entailed by the trace and the point in time. To satisfy boolean conjunction, both conjuncts need to be satisfied. Therefore, both conjuncts are checked separately for satisfaction given the trace and the point in time. For the *next* operator, we check whether the formula following the operator is fulfilled one time step later at $t+1$. An *until* formula $\varphi \mathcal{U} \psi$ is satisfied by a trace tr and a time point t, iff at some point in time ψ is satisfied and until this time point φ is satisfied.

A trace tr satisfies a formula φ iff $tr, 0 \models \varphi$.

2.2 HyperLTL

Hyperproperties are sets of sets of traces. While temporal logics describe sets of traces, we need temporal hyperlogics to describe sets of sets of traces. Hyperlogics can relate traces and specify behavior of traces with respect to other traces. They are thus more powerful than standard temporal logics. The temporal logic LTL can only describe the behavior of an individual trace. HyperLTL is a widely used hyperlogic that extends LTL with explicit quantification over traces. Explicit quantification over traces allows referring to specific traces in the formula and using indexed atomic propositions to determine what trace an atomic proposition comes from. In the following, we have a look at HyperLTL as introduced by Clarkson et al. [2].

2.2.1 Syntax

Definition 2.4 (Syntax)
A HyperLTL formula is constructed using the following grammar:

$$\varphi \coloneqq \exists \pi.\varphi \mid \forall \pi.\varphi \mid \psi$$
$$\psi \coloneqq a_\pi \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \mathcal{U} \psi$$

In the indexed atomic proposition a_π, the index π is a trace variable from the set of traces variables $\mathcal{V}$.
The syntax allows a sequence of quantifiers of arbitrary length before the formula described by the second category of the syntax starts. All atomic propositions occurring in the formula are indexed by trace variables indicating from which trace they stem. A formula is *closed* if all trace variables that occur as indices in the formula were introduced by a quantifier in the prefix. We have the same boolean and temporal operators as for LTL.

Example 2.2 (Observational Determinism)
Observational Determinism [8] is a popular hyperproperty stating that if on two traces the observable inputs are the same, then the observable outputs must also be the same. Using HyperLTL, it is expressed by the formula

$$\forall\pi\forall\pi'.\,\square\,(i_\pi \leftrightarrow i_{\pi'}) \rightarrow \square\,(o_\pi \leftrightarrow o_{\pi'})$$

The indexed atomic propositions i_π and $i_{\pi'}$ are the observable inputs from the traces π and π' respectively and o_π and $o_{\pi'}$ are the observable outputs from the traces π and π', respectively.

2.2.2 Semantics

Let T be a set of traces. Let $\Pi : \mathcal{V} \rightarrow (2^{AP})^\omega$ be a trace assignment that maps trace variables to traces. $\Pi_\emptyset$ denotes the empty assignment. $\Pi[\pi \rightarrow tr]$ denotes the assignment that is the same as Π except that the trace variable π now maps to trace tr. $\Pi(\pi)$ denotes the trace that is assigned to trace variable π in Π. $\Pi(\pi)[i, \infty]$ denotes the trace that is assigned to π in Π starting at position i. The satisfaction of a HyperLTL formula is evaluated with respect to a set of traces T, a trace assignment Π and a time point $t \in \mathbb{N}$ as follows:

Definition 2.5 (Semantics)

$$\begin{aligned}
T, \Pi, t &\models a_\pi && \Leftrightarrow\ a \in \Pi(\pi)(t)\\
T, \Pi, t &\models \neg\psi && \Leftrightarrow\ T, \Pi, t \not\models \psi\\
T, \Pi, t &\models \psi_1 \wedge \psi_2 && \Leftrightarrow\ T, \Pi, t \models \psi_1\ \wedge\ T, \Pi, t \models \psi_2\\
T, \Pi, t &\models \bigcirc\psi && \Leftrightarrow\ T, \Pi, t+1 \models \psi\\
T, \Pi, t &\models \psi_1\,\mathcal{U}\,\psi_2 && \Leftrightarrow\ \exists i \geq t.\ T, \Pi, i \models \psi_2\ \wedge\ \forall t \leq j < i.\ T, \Pi, j \models \psi_1\\
T, \Pi, t &\models \exists\pi.\varphi && \Leftrightarrow\ \exists tr \in T.\ T, \Pi[\pi \rightarrow tr], t \models \psi\\
T, \Pi, t &\models \forall\pi.\varphi && \Leftrightarrow\ \forall tr \in T.\ T, \Pi[\pi \rightarrow tr], t \models \psi
\end{aligned}$$

In order to tell if an atomic proposition a_π is entailed by the given trace set and trace assignment at time point t, we must check whether it is contained in the set of atomic propositions that results when looking up time point t in the trace that the proposition stems from. This trace is obtained by looking up the trace variable π in the trace assignment Π. For the boolean and temporal operators, we proceed as with LTL until at some point we reach an indexed atomic proposition which we look up, as previously described.
If we want to check whether a formula $\exists\pi.\varphi$ that starts with an existential quantifier is satisfied by the given trace set and trace assignment, we must check if there exists a trace in the trace set T such that if we map the quantified trace variable π to that trace, the formula is satisfied by the resulting trace assignment and the trace set. If

we want to check whether a formula $\forall\pi.\varphi$ that starts with a universal quantifier is satisfied by the given trace set and trace assignment, we must check, if for each trace in the trace set T, it holds that if we map the quantified trace variable π to that trace, the formula is satisfied by the resulting trace assignment and the trace set.

A set of traces T satisfies a HyperLTL formula φ, denoted by $T \models \varphi$, iff $T, \Pi_\emptyset, 0 \models \varphi$.

2.3 HyperLTL Synthesis

In HyperLTL synthesis [11], the goal is to automatically construct a system that fulfills a given specification expressed as a HyperLTL formula. This system is then guaranteed to fulfill the expressed hyperproperties by construction. In order to construct such a system, we need to construct a strategy that transforms inputs to system outputs and satisfies the specification. Since a system transforms inputs to outputs, a strategy $f : (2^I)^+ \to (2^O)$, which is a mapping of sequences of sets of inputs to a set of outputs, describes how a system reacts to inputs. It is represented by an infinite tree, the so-called *strategy tree* f, that branches over all possible inputs and has nodes labeled with the outputs.

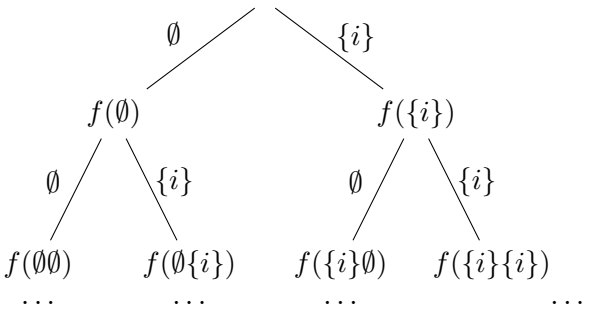

Figure 2.2: A strategy tree for a strategy $f : (2^I)^+ \to (2^O)$ [7]

This means, given an input sequence $w \in (2^I)^+$ the corresponding output $f(w)$ is the label of the node in the tree that is reached when branching as described by the input. If we collect all inputs and outputs along a path, we obtain a trace. Formally, the trace corresponding to the infinite input sequence $w = w_0 w_1 \ldots \in (2^I)^\omega$ is defined as $(w_0 \cup f(w_0))\ (w_1 \cup f(w_0 w_1))(w_2 \cup f(w_0 w_1 w_2))\ \ldots \in (2^{I \cup O})^\omega$.
We lift the operator for set containment $\in$ to the containment of a trace in a strategy. A trace $v = v_0 v_1 v_2 \ldots \in (2^{I \cup O})^\omega$ is contained in a strategy tree induced by a strategy $f : (2^I)^+ \to (2^O)$ iff $f((v_0 \cap I) \ldots (v_i \cap I)) = v_i \cap O$ for all $i \geq 0$.
We call the set that results from collecting all the traces in a tree *traces*(f) defined as $\{v \mid v \in f\}$. While for LTL synthesis it suffices that each trace from this set fulfills the specified formula individually, for HyperLTL synthesis the traces from this set

must be in a specified relationship with each other.
If the set $traces(f)$ resulting from the strategy tree f satisfies the formula, we say that the strategy f realizes the specification. A formula is realizable if and only if there exists a strategy that realizes the specification. Following the definition by Finkbeiner et al. [11], we formally define HyperLTL realizability:

Definition 2.6 (HyperLTL Realizability)
A HyperLTL formula φ over atomic propositions $AP = I \cup O$ is *realizable* if there is a strategy $f : (2^I)^+ \to (2^O)$ that satisfies φ, that is $traces(f) \models \varphi$.

The HyperLTL realizability problem is in general undecidable. However, certain decidable fragments can be characterized, as done by Finkbeiner et al. [11]. They classify the decidability results based on the structure of the quantifier prefix. The realizability problem of a formula that contains only one universal quantifier is equivalent to the LTL realizability problem and, thus, decidable. Once there is more than one universal quantifier, the problem becomes in general undecidable. Consequently, both the universal $\forall^*$ fragment and the $\forall^*\exists^*$ fragment are undecidable as well. However, the realizability of the linear $\forall^*$ fragment, i.e., all universal HyperLTL formulas that can be represented by an equivalent formula that have a distributed form and contain only two universal quantifiers, is decidable. The existential fragment $\exists^*$, meaning a sequence of arbitrary length of existential quantifiers, is PSPACE-complete. If the existential fragment is extended with one universal quantifier $\exists^*\forall^1$, it is still decidable and lies within 3EXPTIME.

Finkbeiner et al. present an approach for bounded realizability of universal HyperLTL formulas [11]. They also implement a bounded approach for finding counterexamples if the strategy is unrealizable. Based on these two approaches, they implement BoSyHyper, a prototype synthesis tool for hyperproperties expressed in universal HyperLTL. The bounded approach and the tool were later extended to HyperLTL formulas with one quantifier alternation by Coenen et al. [3]. BoSyHyper constructs a strategy using a bounded synthesis algorithm if the formula is realizable and provides a counterexample if the formula is unrealizable. Bounded synthesis [7] is an instance of synthesis that focuses on finding small implementations. The idea behind this approach is based on the fact that in practice, realizable specifications often have reasonably small implementations. Bounded synthesis uses a precomputed maximal bound on the number of states and only considers implementations with numbers of states up to this bound. If up to this bound no implementation was found, then with this bound none exists. BoSyHyper is based on the LTL tool for bounded synthesis BoSy [11]. BoSyHyper splits the formula into an LTL formula and a part that contains the hyperproperty in the shape of a HyperLTL formula. Due to this separation, two constraint systems are created that are more concise than if it had been one. The constraint system of the LTL formula can be created using BoSy and the constraint system resulting from the hyperproperty is constructed using the bounded synthesis approach for hyperproperties.

2.4 TSL

In this section, we introduce the temporal stream logic TSL as defined by Finkbeiner et al. [10]. We introduce the necessary definitions first. A value can be of arbitrary type and stems from the set $\mathscr{V}$ of all values. $\mathscr{V}$ subsumes the boolean values, i.e., $\mathcal{B} \subseteq \mathscr{V}$. We use streams to express the constant flow of data over time. A stream $s : \mathbb{N} \to \mathscr{V}$ maps a point in time to a value. We use a stream for each data component we want to represent. If there is for instance data from two different input sources, each source is represented by an input stream. Apart from input streams, there are output streams and computation streams, at which we have a more detailed look later. A function $f : \mathscr{V}^n \to \mathscr{V}$ computes a new value from n given values and stems from the set of all functions of arbitrary arity denoted by $\mathcal{F}$. Constants are expressed through functions of arity 0 and stem from the set $\mathcal{F} \cap \mathscr{V}$ since they are values as well. A predicate $p : \mathscr{V}^n \to \mathcal{B}$ checks a property over n values. The set of all predicates of arbitrary arity is denoted by $\mathcal{P}$ and is a subset of $\mathcal{F}$, i.e., $\mathcal{P} \subseteq \mathcal{F}$.

A TSL formula describes properties on a reactive system that, given a finite number of inputs $\mathbb{I}$, produces a finite number of outputs $\mathbb{O}$ in every time step t. The inputs are obtained from an infinite input stream, that for each time step provides new inputs. Cells $\mathbb{C}$ are used to store values during the computation making them reusable as inputs in the respective next step. The system uses functions without side-effects to transform the values of the input streams in every time step and either pass them on to an infinite output stream or to a cell where it can serve as input for, for instance, another function computation.

In TSL there are functions $f \in \mathcal{F}$ and their compositions used to transform values, and predicates $p \in \mathcal{P}$ that control the flow of data in the system. Both functions and predicates are represented through a term-based notation.

Definition 2.7 (Term Notation)

Function Term:

$$\tau_F \coloneqq \mathtt{s_i} \mid (\mathtt{f}\ \tau_F^0\ \tau_F^1\ \ldots\ \tau_F^{n-1})$$

Predicate Term:

$$\tau_P \coloneqq\ \mathtt{p}\ (\tau_F^0\ \tau_F^1\ \ldots\ \tau_F^{n-1})$$

Update: $[\![\mathtt{s_o} \leftarrowtail \tau_F]\!]$

A function term τ_F can be obtained from an input or a cell value $\mathtt{s_i} \in \mathbb{I} \cup \mathbb{C}$ or it can be a function application recursively applied to function terms. A predicate term τ_P is a predicate recursively applied to function terms. An update is either a function term stored in a cell or passed on to an output $\mathtt{s_o} \in \mathbb{C} \cup \mathbb{O}$. The sets of function and predicate terms and updates are denoted by $\mathcal{T}_F, \mathcal{T}_P$, and $\mathcal{T}_U$ respectively, and $\mathcal{T}_P \subseteq \mathcal{T}_F$.

We use $\mathtt{s_i}, \mathtt{s_o}, \mathtt{f}$, and $\mathtt{p}$ as symbolic representations of inputs and cells, outputs and cells, functions, and predicates respectively. Furthermore, TSL uses uninterpreted functions and predicates. Consequently, functions and predicates are not tied to a specific implementation. We use $\mathbb{F}$ to denote the set of function literals and $\mathbb{P}$ to denote the set of predicate literals. Literals like $\mathtt{s_i}, \mathtt{s_o}, \mathtt{f}$, and $\mathtt{p}$ are used to construct terms as described in the term notation. Again it holds that $\mathbb{P} \subseteq \mathbb{F}$. In order to give meaning to the uninterpreted representation, we use an evaluation function $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$. Note that, since $\mathbb{P} \subseteq \mathbb{F}$ and $\mathcal{P} \subseteq \mathcal{F}$, the evaluation function also provides, though not explicitly stated, a semantic interpretation for predicate literals. Terms can be compared syntactically with the equivalence relation $\equiv$.

To describe the reactive system more precisely, we have a closer look at input and output streams. A momentary input $i \in \mathbb{I} \to \mathscr{V}$ assigns inputs $\mathtt{i} \in \mathbb{I}$ to values $v \in \mathscr{V}$ and thus describes what value the input has. In the following, we denote $\mathbb{I} \to \mathscr{V}$ by $\mathcal{I}$ for readability reasons. Input streams are infinite streams that consist of an infinite sequence of momentary inputs $\iota \in \mathcal{I}^\omega$, one per point in time. Similarly, we define momentary outputs $o \in \mathbb{O} \to \mathscr{V}$ to assign a value $v \in \mathscr{V}$ to each output $\mathtt{o} \in \mathbb{O}$, where from now on $\mathcal{O} = \mathbb{O} \to \mathscr{V}$. Output streams $\rho \in \mathcal{O}^\omega$ are infinite streams that consist of an infinite sequence of momentary outputs. In order to reason about how the input streams are transformed to output streams, we use the notion of a computation σ, which is used to describe the behavior of cells. Capturing the behavior of each cell describes the behavior of the entire system since every function transformation is stored in a cell and those are only performed when the predicates guided the control to do so. Since TSL abstracts from data, it also abstracts from actually computing the values of function applications, allowing us to specify a computation without fixing a semantics for function literals. A computation thus only determines the function terms that are used to compute outputs and cell updates. A computation is composed of computation steps $c \in (\mathbb{O} \cup \mathbb{C}) \to \mathcal{T}_F$ that assign cells $\mathtt{s_o} \in (\mathbb{O} \cup \mathbb{C})$ to function terms $\tau_F \in \mathcal{T}_F$. Let $\mathcal{C} = (\mathbb{O} \cup \mathbb{C}) \to \mathcal{T}_F$. A computation step describes the behavior of the system with regard to the control flow at a single point in time. Since a computation $\sigma \in \mathcal{C}^\omega$ is used to transform an infinite input stream to an infinite output stream, it is as well an infinite sequence of computation steps where each step transforms a momentary input to a momentary output. However, in order to obtain an output stream, we need a concrete interpretation for functions and predicates in addition to the input stream. That is because a momentary output assigns outputs to concrete values that stem from function terms. Thus, in order to get a concrete value, we need to evaluate predicates and functions. As already stated, let $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$ be some function evaluation. We assume that every cell $\mathtt{c} \in \mathbb{C}$ contains some initial value for each stream at the initial point in time expressed through a predefined constant $init_c \in \mathcal{F} \cap \mathscr{V}$.

In order to obtain a concrete value for a function term at a certain point in time t, we use the evaluation function $\eta_{\langle\cdot\rangle} : \mathcal{C}^\omega \times \mathcal{I}^\omega \times \mathbb{N} \times \mathcal{T}_F \to \mathscr{V}$:

Definition 2.8 (Evaluation Function η)

$$\eta_{\langle\cdot\rangle}(\sigma, \iota, t, \mathtt{s_i}) = \begin{cases} \iota(t)(\mathtt{s_i}) & \text{if } \mathtt{s_i} \in \mathbb{I} \\ init_{\mathtt{s_i}} & \text{if } \mathtt{s_i} \in \mathbb{C} \wedge t = 0 \\ \eta_{\langle\cdot\rangle}(\sigma, \iota, t-1, \sigma(t-1)(\mathtt{s_i})) & \text{if } \mathtt{s_i} \in \mathbb{C} \wedge t > 0 \end{cases}$$

$$\eta_{\langle\cdot\rangle}(\sigma, \iota, t, \mathtt{f}\ \tau_0 \ldots \tau_{m-1}) = \langle \mathtt{f} \rangle\ \eta_{\langle\cdot\rangle}(\sigma, \iota, t, \tau_0)\ \ldots\ \eta_{\langle\cdot\rangle}(\sigma, \iota, t, \tau_{m-1})$$

As we can see, the evaluation function $\eta_{\langle\cdot\rangle}$ takes a computation $\sigma \in \mathcal{C}^\omega$, an input stream $\iota \in \mathcal{I}^\omega$, a time point $t \in \mathbb{N}$, and a function term $\tau_F \in \mathcal{T}_F$ as arguments and computes the value that the function term evaluates to at point t under σ and ι. Since a function term can either be constructed from inputs, cells, or functions recursively applied to function terms, we get four different cases. The function term τ_F can be of the form $\mathtt{s_i}$ and is then either constructed from inputs or cells. If it stems from the set of inputs $\mathbb{I}$, then we simply look at the value assigned to $\mathtt{s_i}$ by the input stream at time t. If $\mathtt{s_i}$ is constructed from a cell, then we need to make yet another case distinction with regard to the time point t.
If $t = 0$, we are at the initial time point of the computation and there are not yet any values to have been stored in the cell, meaning that it still contains its initially assigned value $init_{\mathtt{s_i}}$.
In the other case where $t > 0$, we know that time has already passed and that possibly new function terms were stored in cell $\mathtt{s_i}$. Therefore, we need to "trace back" how the current content was constructed. This is done by recursively computing the result of the evaluation function from one time step before and replacing $\mathtt{s_i}$ by the function term that is mapped to $\mathtt{s_i}$ by the computation one step before. This computation terminates since t decreases in each step and will eventually reach 0 so that the second case will be entered. The function term τ_F can also be a function application recursively applied to a set of function terms. If we want to evaluate a function term where a function literal is applied to a set of function terms, we do that by evaluating the literal and applying it to its evaluated arguments. Note that, since $\mathbb{P} \subseteq \mathbb{F}$ and $\mathcal{T}_P \subseteq \mathcal{T}_F$, this case covers predicate terms.

Using the evaluation function $\eta_{\langle\cdot\rangle}$, we can construct the concrete output stream $\rho_{\langle\cdot\rangle,\sigma,\iota} \in \mathcal{O}^\omega$ of a system:

$$\rho_{\langle\cdot\rangle,\sigma,\iota}(t)(\mathtt{o}) = \eta_{\langle\cdot\rangle}(\sigma, \iota, t, \sigma(t)(\mathtt{o})), \text{ for all } t \in \mathbb{N}, \mathtt{o} \in \mathbb{O}.$$

Output streams map points in time to momentary outputs. Thus, if we provide the output stream with a time point t and an output $\mathtt{o}$, we receive the value that the output has at that point. This value is computed through evaluating the function term that the computation maps to the output at time point t.

2.4.1 Syntax

We now have a look at how to construct formulas using TSL.

Definition 2.9 (Syntax)
A TSL formula φ is generated using the following grammar:

$$\varphi \quad := \quad \tau \in \mathcal{T}_P \cup \mathcal{T}_U \;\mid\; \neg\varphi \;\mid\; \varphi \wedge \varphi \;\mid\; \bigcirc\varphi \;\mid\; \varphi \,\mathcal{U}\, \varphi$$

Atomic components, represented by τ are either a predicate term or an update that expresses the control flow at the respective time point. Furthermore, we have the standard boolean operators, *negation* and *conjunction*, and the standard temporal operators *next* and *until*. Further boolean and temporal operators can be derived like for LTL.

2.4.2 Semantics

Let $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$ be an evaluation function, $\iota \in \mathcal{I}^\omega$ an input stream, and $\sigma \in \mathcal{C}^\omega$ a computation. Then the satisfaction of a TSL formula φ w.r.t. to a computation σ and an input stream ι is defined inductively over $t \in \mathbb{N}$ via the following semantics:

Definition 2.10 (Semantics)

$$\begin{aligned}
&\sigma, \iota, t \models_{\langle\cdot\rangle} \mathtt{p}\ \tau_0 \ldots \tau_n && \Leftrightarrow\ \eta_{\langle\cdot\rangle}(\sigma, \iota, t, \mathtt{p}\ \tau_0 \ldots \tau_n)\\
&\sigma, \iota, t \models_{\langle\cdot\rangle} [\![\mathtt{s} \leftarrowtail \tau]\!] && \Leftrightarrow\ \sigma(t)(\mathtt{s}) \equiv \tau\\
&\sigma, \iota, t \models_{\langle\cdot\rangle} \neg\varphi && \Leftrightarrow\ \sigma, \iota, t \not\models_{\langle\cdot\rangle} \varphi\\
&\sigma, \iota, t \models_{\langle\cdot\rangle} \varphi_1 \wedge \varphi_2 && \Leftrightarrow\ \sigma, \iota, t \models_{\langle\cdot\rangle} \varphi_1 \ \wedge\ \sigma, \iota, t \models_{\langle\cdot\rangle} \varphi_2\\
&\sigma, \iota, t \models_{\langle\cdot\rangle} \bigcirc\varphi && \Leftrightarrow\ \sigma, \iota, t+1 \models_{\langle\cdot\rangle} \varphi\\
&\sigma, \iota, t \models_{\langle\cdot\rangle} \varphi_1 \,\mathcal{U}\, \varphi_2 && \Leftrightarrow\ \exists i \geq t.\ \sigma, \iota, i \models_{\langle\cdot\rangle} \varphi_2 \ \wedge\ \forall t \leq j < i.\ \sigma, \iota, j \models_{\langle\cdot\rangle} \varphi_1
\end{aligned}$$

In order to evaluate a predicate at a certain time point, we use the evaluation function and compute whether the predicate holds on the given function terms. We use the evaluation function since we want to know the concrete boolean value that the predicate evaluates to.
In contrast, when evaluating an update, we can stay on the symbolic level since we just want to check syntactically whether the control flow of the system was as expected. Since the computation σ captures the behavior of the cells and thus describes the control flow, we syntactically compare the function term that we want to update the cell $\mathtt{s}$ with, with the function term that we find for $\mathtt{s}$ at time point t in the computation using the equivalence relation $\equiv$.

In order to check whether $\neg\varphi$ is satisfied by a computation and an input stream, we check whether the formula φ is not entailed by the computation and the input stream. The boolean and temporal operators are evaluated as for LTL.
A computation σ and an input stream ι satisfy a TSL formula φ denoted by $\sigma, \iota \models_{\langle\cdot\rangle} \varphi$ iff $\sigma, \iota, 0 \models_{\langle\cdot\rangle} \varphi$.

Chapter 3

HyperTSL

In this chapter, we present a hyperlogic for TSL called HyperTSL. Like other hyperlogics, it allows explicit quantification over traces and indexes atomic components in order to specify to which trace they belong. There are two major design decisions that need to be taken when constructing HyperTSL from TSL. The first one is concerned with the function evaluation $\langle\cdot\rangle$ and whether it can be chosen for each trace individually or whether it must be the same for all traces. For HyperTSL, we decide to use the same evaluation function for all traces, as the traces all stem from the same system. As predicates are a subset of functions, we also use the same evaluation function for predicates on all traces. The second design decision for HyperTSL is concerned with how terms are indexed. There are different options for indexing terms with trace variables, as unlike in LTL, the basic components of a TSL formula are not atomic. The basic components of a TSL formula are predicate terms and updates which each consist of several components. In a HyperTSL formula, we can also have multiple predicate terms and updates that stem from different traces. We thus need to find an appropriate way of indexing predicate terms and updates with trace variables. For predicates terms, we can see the predicate literal and its arguments as one atomic component or we can see the arguments as individual components. If we choose to see the predicate literal and its arguments as one atomic component, then we index the entire predicate term with a trace variable which means that all arguments come from this trace. If we choose to see the predicate and its arguments as individual components, we must index each argument individually. This allows predicates to take arguments from different traces. This is dramatically more expressive. A similar approach for updates, where a cell and the value that it is assigned are seen as individual components and thus indexed differently, can however not be taken. Indexing the components of an update individually would allow expressing hyperproperties, where we could assign a value from one trace to the cell of another trace. This would describe behavior that is neither desirable for the hyperlogic nor allowed in TSL. As indexing the components of an update individually entails undesired behavior, the only option is to see updates as atomic components and index them accordingly. As there

is only one possibility for indexing updates and two possibilities for predicate terms, we can derive a hyperlogic and identify a syntactic fragment of it. Depending on what approach is chosen for predicate terms, which immediately affects function terms as well, hyperlogics with different expressivity result. We choose the approach where we index each argument of a predicate term individually for HyperTSL. Choosing the approach where predicate terms are atomic components, results in a fragment of HyperTSL, which we call HyperTSL$^-$ and which we explore further in Section 3.3.

We introduce an indexed term notation for HyperTSL. The indices used in a HyperTSL formula stem from the set of trace variables $\mathcal{V}$. The set of indexed predicate terms $\mathcal{T}_{P,\mathcal{V}}$, and the set of indexed updates $\mathcal{T}_{U,\mathcal{V}}$ are a subset of the set of all indexed terms $\mathcal{T}_{\mathcal{V}}$. Furthermore, it holds that since predicate terms are a subset of function terms in TSL, indexed predicate terms are a subset of indexed function terms in HyperTSL, $\mathcal{T}_{P,\mathcal{V}} \subseteq \mathcal{T}_{F,\mathcal{V}}$. Indexed terms $\tau_\pi \in \mathcal{T}_{\mathcal{V}}$, where $\pi \in \mathcal{V}$, are constructed using the following term-based notation:

Definition 3.1 (Indexed Terms)

Indexed Function Term:

$$\tau_{F,\pi} \coloneqq \mathtt{s}_{\mathtt{i}\pi} \mid \mathtt{f}\ (\tau^0_{F,\pi}\ \tau^1_{F,\pi}\ \ldots\ \tau^{n-1}_{F,\pi})$$

Indexed Predicate Term:

$$\tau_{P,\pi} \coloneqq\ \mathtt{p}\ (\tau^0_{F,\pi}\ \tau^1_{F,\pi}\ \ldots\ \tau^{n-1}_{F,\pi})$$

Indexed Update: $[\![\mathtt{s}_\mathtt{o} \leftharpoondown \tau_{F,\pi}]\!]_\pi$

Where we restrict that all components of an indexed function term occurring in an update must be from the same trace as the update itself.

In an indexed term τ_π, $\pi \in \mathcal{V}$ is the trace variable that indicates the trace that the term stems from. Indexed function terms are either an indexed input or a cell, or constructed from functions that are recursively applied to a set of indexed function terms. Indexed predicate terms are predicates applied to a set of indexed function terms. The definition of indexed function and predicate terms syntactically allows function computations and predicate evaluations to take arguments from different traces. We further explore this feature when defining the semantics. Indexed updates $[\![\mathtt{s}_\mathtt{o} \leftharpoondown \tau_{F,\pi}]\!]_\pi$ are updates indexed with a trace variable π but with the restriction that all indexed function terms occurring in the indexed function term $\tau_{F,\pi}$ must be indexed by the same trace variable π as the update itself. This syntactically enforces that all indexed function terms in an update stem from the same trace as the update itself. Therefore, a cell can only store function terms that stem from the same execution. At first sight, it might seem as if it would suffice to only use non-indexed function terms for HyperTSL as function terms in an update must be from the same trace as the

update. However, as predicate terms are a subset of function terms, predicates can also occur in an update, for instance $[\![\mathtt{c} \leftarrowtail \mathtt{p}\ (\mathtt{x}_\pi, \mathtt{y}_\pi)]\!]$. Therefore, in order to comply with the definition of indexed predicate terms, we must use *indexed* function terms in an update and cannot just use standard function terms, even though all components must be from the same trace. Furthermore, indexed function terms can occur in indexed predicate terms. Recall that since $\mathcal{T}_{P,\pi} \subseteq \mathcal{T}_{F,\pi}$, we can also assign predicate terms to cells in updates and we can nest predicate terms though not explicitly stated. All the imposed restrictions on function terms, thus, also hold for predicate terms in the same contexts. We omit parentheses for predicates and functions with only one argument. Next, we define the syntax of HyperTSL. A HyperTSL formula is generated using the following grammar:

Definition 3.2 (Syntax)

$$\varphi \coloneqq \exists\pi.\varphi \mid \forall\pi.\varphi \mid \psi$$
$$\psi \coloneqq \tau_\pi \in \mathcal{T}_{P,\mathcal{V}} \cup \mathcal{T}_{U,\mathcal{V}} \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc\psi \mid \psi\,\mathcal{U}\,\psi$$

The function term τ_π ranges over the set $\mathcal{T}_{P,\mathcal{V}}$ of indexed predicate terms or $\mathcal{T}_{U,\mathcal{V}}$ of indexed updates and the trace variable π ranges over the set of trace variables $\mathcal{V}$. Indexed predicate terms and indexed updates evaluate to *true* or *false*. Note that τ_π cannot be an indexed function term from the set $\mathcal{T}_{F,\mathcal{V}} \setminus \mathcal{T}_{P,\mathcal{V}}$ since the components of a HyperTSL formula must evaluate to a truth value. A HyperTSL formula starts with a sequence of quantifiers of arbitrary length, containing existential or universal quantifiers, followed by a formula from the second category of the grammar. Like for TSL, we have the standard boolean connectives, i.e., negation, and conjunction, as well as the temporal operators *next* and *until*. Other boolean and temporal operators can be derived as for LTL. HyperTSL subsumes TSL, as we can build a formula with only one universal quantifier and then choose the last option in the first category of the grammar, which is equivalent to a TSL formula.
A HyperTSL formula is *closed* if all trace variables that occur as indices in the formula were introduced by a quantifier in the prefix. In the following, we only consider closed formulas.

3.1 Traces

Hyperproperties are evaluated over a set of traces. In order to be able to evaluate a hyperproperty expressed in HyperTSL on a system, we must first specify the traces on which the hyperproperties are evaluated. Intuitively, a trace is a sequence that for each point in time provides a momentary insight into the system behavior. For LTL, the momentary internal state of a system is expressed through a set of atomic propositions, which encode both the momentary input and output of the system. As both inputs and outputs are represented by atomic propositions, an LTL trace stems from the set

of all traces over AP, $(2^{AP})^\omega$ (see Figure 3.1). In order to be able to represent the momentary internal state of a system specified in TSL, we need both the momentary inputs and momentary outputs. While the inputs can be accessed through the input stream, it is not so easy to obtain the outputs since we need to fix a function evaluation that gives meaning to the symbolic representation of the functions and predicates used to compute the outputs. Since we do not want to fix a function evaluation this early, we cannot use the output stream to depict the behavior of the system. Another way to depict the behavior of the system is through a computation. A computation captures the behavior of each cell and, thus, depicts the control flow and the behavior of the system for each point in time. An input stream and a computation suffice to compute an output stream once a concrete function evaluation is chosen. Therefore, an input stream and a computation suffice to depict the momentary internal state of a system execution. Using only the computation and the input stream to describe an execution trace leaves enough room for choosing whatever function evaluation fits our purposes best since, after all, the function evaluation in combination with the control flow is what makes the system realize the desired behavior. A HyperTSL trace is thus a pair (σ, ι) of a computation $\sigma \in \mathcal{C}^\omega$ and an input stream $\iota \in \mathcal{I}^\omega$. For comparison, Figure 3.1 illustrates what a HyperLTL trace π_L and a trace π in HyperTSL look like.

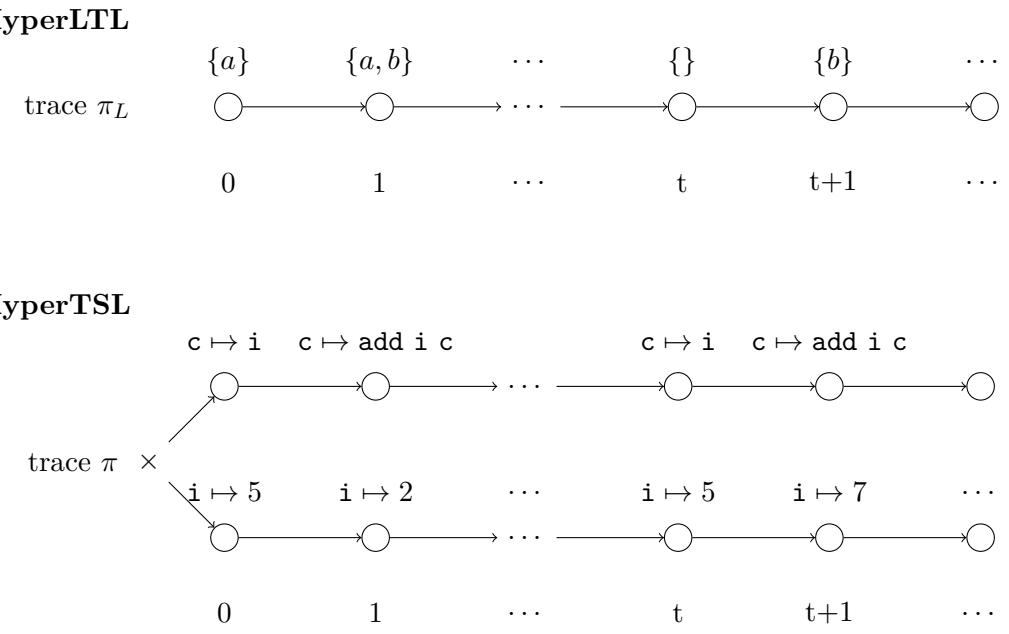

Figure 3.1: A HyperLTL trace and a HyperTSL trace

On the HyperLTL trace π_L, the atomic proposition a holds in the first time step, a and b hold in the second time step, no atomic proposition holds in time step t and

b holds again in $t+1$. On the HyperTSL trace π the input stream assigns 5 to the input `i` and the computation assigns the input `i` to the cell `c` in the first time step. This means that in the first time step, cell `c` stores the value of `i`, which is 5. In the second time step, the input stream provides the value 2 for `i` and cell `c` stores the result of adding the current input value 2 and the stored value 5. In time steps t and $t+1$ the behavior is similar to what we just described.
We fix the type of the HyperTSL trace assignment that maps trace variables to traces as $\Pi : \mathcal{V} \to (\mathcal{C}^\omega \times \mathcal{I}^\omega)$. Let #1 and #2 be the projections that return the first and the second component of a pair respectively.
In order to evaluate indexed terms, we need an evaluation function to take care of the potentially different traces, that the values stem from. Consider the predicate term $\mathtt{p}\ (\mathtt{f}(\mathtt{x}_\pi, \mathtt{x}_{\pi'}))$. If we tried to evaluate this term using the evaluation function $\eta_{\langle\cdot\rangle}$ we would need to commit to one trace before unfolding the definition of $\eta_{\langle\cdot\rangle}$ for predicate terms. Then we could unfold the definition of $\eta_{\langle\cdot\rangle}$ for function terms, but when evaluating the arguments of the function term, $\mathtt{x}_\pi$ and $\mathtt{x}_{\pi'}$, we encounter the problem that we are already committed to a specific trace, while the arguments come from two potentially different traces. We define an evaluation function $\mu_{\langle\cdot\rangle} : (\mathcal{V} \to (\mathcal{C}^\omega \times \mathcal{I}^\omega)) \times \mathbb{N} \times \mathcal{T}_\mathcal{V} \to \mathcal{V}$ that given a trace assignment Π and a point in time t evaluates an indexed term τ_π.

Definition 3.3 (Evaluation Function $\mu_{\langle\cdot\rangle}$ for HyperTSL)

$$\mu_{\langle\cdot\rangle}(\Pi, t, \mathtt{s}_{\mathtt{i},\pi}) = \begin{cases} \#2(\Pi(\pi))(t)(\mathtt{s}_\mathtt{i}) & \text{if } \mathtt{s}_\mathtt{i} \in \mathbb{I} \\ init_{\mathtt{s}_\mathtt{i}} & \text{if } \mathtt{s}_\mathtt{i} \in \mathbb{C} \ \wedge\ t = 0 \\ \mu_{\langle\cdot\rangle}(\Pi, t-1, \mathtt{index}\ \pi\ (\#1(\Pi(\pi))(t-1)(\mathtt{s}_\mathtt{i}))) & \text{if } \mathtt{s}_\mathtt{i} \in \mathbb{C} \ \wedge\ t > 0 \end{cases}$$

$$\mu_{\langle\cdot\rangle}(\Pi, t, \mathtt{f}\ \tau_{\pi_0} \dots\ \tau_{\pi_{n-1}}) \ =\ \langle \mathtt{f} \rangle\ \mu_{\langle\cdot\rangle}(\Pi, t, \tau_{\pi_0}) \dots \mu_{\langle\cdot\rangle}(\Pi, t, \tau_{\pi_{n-1}})$$

where the function $\mathtt{index} : \mathcal{V} \to \mathcal{T}_F \to \mathcal{T}_{F,\mathcal{V}}$ is defined as:

$$\begin{aligned} &\mathtt{index}\ \pi\ \mathtt{s}_\mathtt{i} && = \mathtt{s}_{\mathtt{i},\pi} \\ &\mathtt{index}\ \pi\ (\mathtt{f}\ \tau_0 \dots\ \tau_{n-1}) && = \mathtt{f}\ (\mathtt{index}\ \pi\ \tau_0) \dots\ (\mathtt{index}\ \pi\ \tau_{n-1}) \end{aligned}$$

As we can see, the evaluation function $\mu_{\langle\cdot\rangle}$ takes a trace assignment Π, a time point $t \in \mathbb{N}$, and an indexed function term $\tau_{F,\pi} \in \mathcal{T}_{F,\mathcal{V}}$ as arguments and computes the value that the indexed function term evaluates to at point t. Since an indexed function term can either be constructed from inputs, cells, or functions recursively applied to function terms, we get four different cases. The indexed function term $\tau_{F,\pi}$ can be of the form $\mathtt{s}_{\mathtt{i},\pi}$ and is then either constructed from inputs or cells. If it stems from the set of inputs $\mathbb{I}$, then we simply look at the value assigned to $\mathtt{s}_\mathtt{i}$ by the input stream of trace π at time t. The input stream that belongs to trace π is obtained by looking up the trace variable π in the trace assignment Π and then projecting on the second

component of the trace. If $\mathtt{s_{i,\pi}}$ is constructed from a cell, then we need to make yet another case distinction with regard to the time point t.
If $t = 0$, we are at the initial time point of the computation and there are not yet any values to have been stored in the cell, meaning that it still contains its initially assigned value $init_{\mathtt{s_i}}$. The initial value of a cell is independent of traces as the execution has not even started, which is why we simply return the initial value of the cell.
In the other case, where $t > 0$, we know that time has already passed and that possibly new function terms were stored in cell $\mathtt{s_i}$. Therefore, we need to "trace back" how the current content was constructed. This is done by recursively computing the result of the evaluation function from one time step before and replacing $\mathtt{s_i}$ by the function term that is mapped to $\mathtt{s_i}$ by the computation one step before. The computation that contains the matching information is obtained by projection on the first component of the trace that is mapped to trace variable π in Π. As the function term that is stored in the cell, can itself be composed of other function terms and is evaluated by $\mu_{\langle\cdot\rangle}$ again, we must index the function term with the trace variable of the trace we are currently on, in order to match the type of the evaluation function. To do so, we use the function `index` which given a trace variable and a function term returns the indexed function term, which results when indexing each argument component of the function term with the given trace variable. Each argument of the function term can be indexed with the same variable because we only allowed function terms in updates to take arguments from the same trace. The indexed function term $\tau_{F,\pi}$ can also be a function application recursively applied to a set of indexed function terms. If we want to evaluate a function term where a function literal is applied to a set of function terms, we do that by evaluating the literal and applying it to its evaluated arguments. Note that, since $\mathbb{P} \subseteq \mathbb{F}$ and $\mathcal{T}_{,\mathcal{V}} \subseteq \mathcal{T}_{F,\mathcal{V}}$, this case covers indexed predicate terms as well. While, conceptually, the evaluation function $\mu_{\langle\cdot\rangle}$ is similar to the evaluation function $\eta_{\langle\cdot\rangle}$ for TSL (see Definition 2.8), the difference is, that $\eta_{\langle\cdot\rangle}$ uses a concrete trace to evaluate a function term, while here, we do not commit to one trace but instead take the entire trace assignment to evaluate a function term. For HyperTSL, we must provide the evaluation function with the trace assignment, as indexed function terms can be nested, and thus arguments of indexed function terms can come from different traces. Therefore, we cannot directly determine what trace is needed to evaluate a component of an indexed function term.
The satisfaction of a HyperTSL formula is evaluated with respect to a trace assignment $\Pi : \mathcal{V} \to (\mathcal{C}^\omega \times \mathcal{I}^\omega)$ and a set of traces T at a time point t as follows:

Definition 3.4 (Semantics)

$$
\begin{aligned}
&\Pi, T, t \models_{\langle\cdot\rangle} \mathtt{p}\ (\tau_{\pi_0}\ \ldots\ \tau_{\pi_{n-1}}) && \Leftrightarrow\ \mu_{\langle\cdot\rangle}(\Pi, t, \mathtt{p}\ (\tau_{\pi_0}\ \ldots\ \tau_{\pi_{n-1}}))\\
&\Pi, T, t \models_{\langle\cdot\rangle} [\![\mathtt{s} \leftarrowtail \tau]\!]_\pi && \Leftrightarrow\ \#1(\Pi(\pi))(t)(\mathtt{s}) \equiv \tau\\
&\Pi, T, t \models_{\langle\cdot\rangle} \neg\varphi && \Leftrightarrow\ \Pi, T, t \not\models_{\langle\cdot\rangle} \varphi\\
&\Pi, T, t \models_{\langle\cdot\rangle} \varphi_1 \wedge \varphi_2 && \Leftrightarrow\ \Pi, T, t \models_{\langle\cdot\rangle} \varphi_1\ \wedge\ \Pi, T, t \models_{\langle\cdot\rangle} \varphi_2\\
&\Pi, T, t \models_{\langle\cdot\rangle} \bigcirc\varphi && \Leftrightarrow\ \Pi, T, t+1 \models_{\langle\cdot\rangle} \varphi\\
&\Pi, T, t \models_{\langle\cdot\rangle} \varphi_1 \mathcal{U} \varphi_2 && \Leftrightarrow\ \exists i \geq t : \Pi, T, i \models_{\langle\cdot\rangle} \varphi_2\ \wedge\ \forall t \leq j < i : \Pi, T, j \models_{\langle\cdot\rangle} \varphi_1\\
&\Pi, T, t \models_{\langle\cdot\rangle} \Diamond\varphi && \Leftrightarrow\ \exists i \geq t : \Pi, T, i \models_{\langle\cdot\rangle} \varphi\\
&\Pi, T, t \models_{\langle\cdot\rangle} \Box\varphi && \Leftrightarrow\ \forall i \geq t : \Pi, T, i \models_{\langle\cdot\rangle} \varphi\\
&\Pi, T, t \models_{\langle\cdot\rangle} \exists\pi.\varphi && \Leftrightarrow \exists(\sigma, \iota) \in T : \Pi[\pi \to (\sigma, \iota)], T, t \models_{\langle\cdot\rangle} \varphi\\
&\Pi, T, t \models_{\langle\cdot\rangle} \forall\pi.\varphi && \Leftrightarrow\ \forall(\sigma, \iota) \in T : \Pi[\pi \to (\sigma, \iota)], T, t \models_{\langle\cdot\rangle} \varphi
\end{aligned}
$$

As already mentioned, predicates can take arguments from different traces, where every argument is indexed by the trace it stems from. Therefore, when evaluating a predicate term under a trace assignment Π and a trace set T at a time point t, we check whether the evaluation function $\mu_{\langle\cdot\rangle}$ evaluates the predicate to true given the trace assignment Π and the time point t. As the arguments of a predicate can be from different traces, it is possible that each argument comes from a different trace. Since the evaluation function $\mu_{\langle\cdot\rangle}$ gets the entire trace assignment, it can evaluate the arguments of the predicate term on the matching traces, as the traces are accessible through the trace assignment. Eventually, the arguments of the predicate term will have been broken down to either a cell or an input stream which must be from *one* specific trace. Therefore the evaluation terminates at some point. For indexed updates, we required that every indexed function term occurring in the update must be indexed by the same trace variable as the update. In order to evaluate an update, we access the computation at time point t and compare the function term that is assigned to cell $\mathtt{s}$ with the function term that is assigned by the update using the syntactic evaluation function $\equiv$. The computation is obtained by looking up the trace variable in the trace assignment. The semantics for boolean and temporal operators is recursively defined as for HyperLTL. When there is an existential quantification over a trace π, then a specific pair (σ, ι) is picked from T and the trace assignment is updated such that the mapping of all trace variables that are not π remain the same, but π now maps to the specific pair. Note that for existential quantification, such a trace *must* exist. When there is a universal quantifier, it must be that all traces from the set fulfill the property. Given a concrete function evaluation $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$, a set of traces T satisfies a formula φ w.r.t. $\langle\cdot\rangle$ iff $\Pi_\emptyset, T, 0 \models_{\langle\cdot\rangle} \varphi$.

3.2 Battery Example

In this example, we illustrate the use of HyperTSL by expressing hyperproperties regarding the behavior of a battery using HyperTSL. We first describe properties of such a battery system using TSL. We consider a reactive system that uses sunlight to charge its battery and that consumes energy depending on the inputs it gets from the input stream `consume`. It needs constant sunlight to charge its battery, expressed through the input stream `sun` and starts to lose energy whenever the sunlight input drops below a certain threshold `t`. Whenever this is the case, the current value of the battery `bat` is decremented by one. The value of the battery can range between 0 and 100, where it is "empty" at 0 and fully charged at 100. Once it is below 15, an `alarm` should go off, signaling low energy of the system. Whenever the sunlight input is above the given threshold, the battery is charging, meaning its value is incremented by one in every time step. If the alarm was on and the battery gets recharged to be above 15, the alarm should be turned off. We describe the system using TSL as follows:

$$\begin{aligned}\mathbb{I} &= \{\texttt{sun},\ \texttt{consume}\}\\ \mathbb{O} &= \{\texttt{bat},\ \texttt{alarm}\}\\ \mathbb{C} &= \{\texttt{bat},\ \texttt{alarm}\}\\ \mathcal{T}_P &= \{\texttt{below15},\ \texttt{less},\ \texttt{greater},\ \texttt{equal},\ \texttt{p}\}\\ \mathcal{T}_F &= \mathcal{T}_P \cup \{\texttt{t},\ \texttt{inc},\ \texttt{dec},\ \texttt{dec2},\ \texttt{on},\ \texttt{off},\ \texttt{min},\ \texttt{max}\}\end{aligned}$$

We use uninterpreted functions and predicates, but we give the function literals and predicate literals a semantics for the example in order to specify understandable properties. We use the functions $\texttt{inc}(\texttt{x})$ and $\texttt{dec}(\texttt{x})$ for incrementation and decrementation respectively. The function $\texttt{dec2}(\texttt{x})$ decrements the value `x` by two instead of the usual one. The functions $\texttt{min}(\texttt{x},\texttt{y})$ and $\texttt{max}(\texttt{x},\texttt{y})$ determine the minimum and maximum of two values `x` and `y` respectively and are used to keep the `bat` values in the range of 0 to 100. In TSL, constants are expressed through zero-ary functions. We use the constant `t` for the fixed threshold and `on` and `off` to describe the state of the alarm. We use the predicate $\texttt{below15}(\texttt{bat})$ to determine whether the battery value is below 15 and the predicates $\texttt{less}(\texttt{x},\texttt{y})$, $\texttt{greater}(\texttt{x},\texttt{y})$, and $\texttt{equal}(\texttt{x},\texttt{y})$ to compare values `x` and `y`. The predicates $\texttt{less},\texttt{greater}$, and `equal` have the expected meaning. Lastly, we have the predicate `p consume` that determines whether energy is consumed because of the input stream `consume`. We can now specify the desired properties.
In order to accurately model the described charging behavior, we must make a case distinction on whether energy is consumed or not, and whether the `sun` value is less, equal, or greater than the threshold. If the `sun` value is greater than the threshold and no energy is consumed, we want the `bat` value to be incremented. If the battery is fully charged, i.e., it is at 100, and still gets energy, the battery value stays at 100.

$$\square\,(\texttt{greater}(\texttt{sun},\texttt{t}) \wedge \neg(\texttt{p consume}) \rightarrow [\![\texttt{bat} \leftarrowtail \texttt{min}(100, \texttt{inc bat})]\!]) \tag{3.1}$$

If the `sun` value is less than the threshold and no energy is consumed, the `bat` value should be decremented. The value of a battery cannot be negative so we use `max` to assure that the value assigned to `bat` is greater or equal to 0.

$$\square\,(\texttt{less}(\texttt{sun},\texttt{t}) \wedge \neg(\texttt{p consume}) \rightarrow [\![\texttt{bat} \leftarrowtail \texttt{max}(0, \texttt{dec bat})]\!]) \tag{3.2}$$

If the `sun` value is equal to the threshold and no energy is consumed, the `bat` value should not change at all.

$$\square\,(\texttt{equal}(\texttt{sun},\texttt{t}) \wedge \neg(\texttt{p consume}) \rightarrow [\![\texttt{bat} \leftarrowtail \texttt{bat}]\!]) \tag{3.3}$$

If the `sun` value is greater than the threshold and energy is consumed, the `bat` value does not change either. This is since the battery would charge because the `greater` predicate holds, but at the same time energy is consumed, which means the `bat` value decreases.

$$\square\,(\texttt{greater}(\texttt{sun},\texttt{t}) \wedge \texttt{p consume} \rightarrow [\![\texttt{bat} \leftarrowtail \texttt{bat}]\!]) \tag{3.4}$$

If the `sun` value is less than the threshold and energy is consumed we must decrement the current `bat` value by two. The value is decremented by two since we need to account for the loss of energy by consumption and by lack of sunshine.

$$\square\,(\texttt{less}(\texttt{sun},\texttt{t}) \wedge \texttt{p consume} \rightarrow [\![\texttt{bat} \leftarrowtail \texttt{max}(0, \texttt{dec2 bat})]\!]) \tag{3.5}$$

If the `sun` value is equal to the threshold and energy is being consumed, we decrement the `bat` value by one.

$$\square\,(\texttt{equal}(\texttt{sun},\texttt{t}) \wedge \texttt{p consume} \rightarrow [\![\texttt{bat} \leftarrowtail \texttt{max}(0, \texttt{dec bat})]\!]) \tag{3.6}$$

An alarm should be turned on if the battery value gets critically low. We check whether this is the case using `below15` and if the predicate evaluates to true, we update the `alarm` cell with `on`, otherwise with `off`. We must cover both cases explicitly, since otherwise, if for one case nothing is specified, the cell could update itself with its current value, $[\![\texttt{alarm} \leftarrowtail \texttt{alarm}]\!]$ and for instance, if `alarm` was previously `on`, it would still be `on` in the next step, even though `bat` was not critically low.

$$\square\,(\texttt{below15 bat} \rightarrow [\![\texttt{alarm} \leftarrowtail \texttt{on}]\!]) \tag{3.7}$$

$$\square\,(\neg\texttt{below15 bat} \rightarrow [\![\texttt{alarm} \leftarrowtail \texttt{off}]\!]) \tag{3.8}$$

Using HyperTSL, we can now specify, for instance, the following hyperproperties on a system that satisfies these TSL properties. We can express determinism of the system using a hyperproperty. Determinism means that for two traces, whenever the inputs are the same, the outputs are also the same. For us, this means, if the inputs provided by `sun` and `consume` are the same on both traces respectively, assuming same initial battery values, then the resulting outputs `alarm` and `bat` are the same as well. We use the uninterpreted predicate `equal` to compare the `sun` and `bat` values from

different traces. This hyperproperty holds on any system that satisfies the above TSL specification.

$$\begin{aligned}\forall\pi\forall\pi'.\square\,(\texttt{equal}(\texttt{sun}_\pi,\texttt{sun}_{\pi'})\ \wedge\ \texttt{p consume}_\pi \leftrightarrow \texttt{p consume}_{\pi'})\\ \rightarrow\square\,([\![\texttt{alarm} \leftarrowtail \texttt{on}]\!]_\pi \leftrightarrow [\![\texttt{alarm} \leftarrowtail \texttt{on}]\!]_{\pi'} \wedge \texttt{equal}(\texttt{bat}_\pi,\texttt{bat}_{\pi'}))\end{aligned} \tag{3.9}$$

Note that if the system had additional secret inputs and some secret output, the above property would state observational determinism (see Example 2.2).
In the next formula, we state that if for a pair of traces that have the same initial battery value and either only consume energy or do not consume energy at all during some predefined time, and they get charged and unloaded in exactly the reverse way for the same amount of time steps each during this predefined time, then the battery values will be equal after these time steps. As an example, the hyperproperty considers 9 time steps.

$$\begin{aligned}\forall\pi\forall\pi'.\Big(&\texttt{equal}(\texttt{bat}_\pi,\texttt{bat}_{\pi'})\ \wedge\ \Big(\bigwedge_{i=0}^{9}(\texttt{p consume}_\pi\ \wedge\ \texttt{p consume}_{\pi'})\\ &\vee\ \bigwedge_{i=0}^{9}(\neg\texttt{p consume}_\pi\ \wedge\ \neg\texttt{p consume}_{\pi'})\Big)\\ &\wedge\ (\bigwedge_{i=0}^{4}\bigcirc_i\ \texttt{greater}(\texttt{sun}_\pi,\texttt{t})\wedge\bigcirc_i\ \texttt{less}(\texttt{sun}_{\pi'},\texttt{t}))\\ &\wedge\ (\bigwedge_{i=5}^{9}\bigcirc_i\ \texttt{greater}(\texttt{sun}_{\pi'},\texttt{t})\wedge\bigcirc_i\ \texttt{less}(\texttt{sun}_\pi,\texttt{t}))\Big)\\ &\rightarrow\ \bigcirc_{10}\ \texttt{equal}(\texttt{bat}_\pi,\texttt{bat}_{\pi'})\end{aligned} \tag{3.10}$$

The notation $\bigcirc_i$ denotes a sequence of i consecutive $\bigcirc$ operators, e.g., $\bigcirc_2\,\varphi$ means $\bigcirc\bigcirc\varphi$. This hyperproperty also holds on a system that fulfills the above TSL properties.
Next, we state that if the sun supply is the same for both systems, while the consumption is different at all times, then it cannot be that the battery is incremented at the same time on both traces.

$$\begin{aligned}\forall\pi\forall\pi'.\square\ \texttt{equal}(\texttt{sun}_\pi,\texttt{sun}_{\pi'})\ \wedge\ \square\,(\texttt{p consume}_\pi\otimes\texttt{p consume}_{\pi'})\\ \rightarrow\square\,(\neg([\![\texttt{bat} \leftarrowtail \texttt{inc bat}]\!]_\pi \wedge [\![\texttt{bat} \leftarrowtail \texttt{inc bat}]\!]_{\pi'}))\end{aligned} \tag{3.11}$$

This hyperproperty also holds on a system that fulfills the above TSL properties.
Furthermore, we can express the hyperproperty that the input provided by `consume` does not influence the battery value. This means that whenever the `sun` values are the same, the battery value should also be the same.

$$\forall\pi\forall\pi'.\square\ \texttt{equal}(\texttt{sun}_\pi,\texttt{sun}_{\pi'})\rightarrow\square\ \texttt{equal}(\texttt{bat}_\pi,\texttt{bat}_{\pi'}) \tag{3.12}$$

This hyperproperty does not hold on the specified system, as the value of `consume` influences the `bat` values as well.
Lastly, we use the fact that updates have a truth value for specifying hyperproperties. If we want to state that whenever the `sun` values are the same, then the `alarm` values also always have the same value, we could use the `equal` predicate for comparing the `alarm` values as in Formula 3.12. But instead, we can check whether `alarm` is updated with `on` in exactly the same time steps on both traces. This check has the same effect as an equality predicate would have since we always either assign `on` or `off` to `alarm`.

$$\forall\pi\forall\pi'.(\square\,\texttt{equal}(\texttt{sun}_\pi, \texttt{sun}_{\pi'})) \rightarrow \square\,([\![\texttt{alarm} \leftarrowtail \texttt{on}]\!]_\pi \leftrightarrow [\![\texttt{alarm} \leftarrowtail \texttt{on}]\!]_{\pi'}) \tag{3.13}$$

The specified hyperproperty does not hold on the system, as like for Formula 3.12 the value of `consume` influences the `bat` value and, thus, also the value stored in the cell `alarm`.

3.3 HyperTSL⁻

HyperTSL⁻ is a fragment of HyperTSL. For HyperTSL we choose to index the arguments of a predicate individually, and for HyperTSL⁻ we choose to index the entire predicate as an atomic component. While indexing the arguments of predicate terms individually allows expressing a wider range of predicate terms, higher expressivity of HyperTSL can cause more challenges compared to HyperTSL⁻ when it comes to problems such as model checking or synthesis. For synthesis, for instance, finding an approximation of HyperTSL⁻ formulas to HyperLTL formulas is more intuitive as the concept of atomic components is more similar for HyperTSL⁻ and HyperLTL than for HyperTSL and HyperLTL, as we see in Chapter 5. Even though HyperTSL⁻ is only a fragment of HyperTSL, it still allows expressing many useful hyperproperties that are not expressible using HyperLTL. We introduce a term notation that realizes the index restriction on predicate terms.

Definition 3.5 (Indexed Term Notation)

Function Term[-]:

$$\tau_F \coloneqq \mathtt{s_i} \mid \mathtt{f}\ (\tau_F^0\ \tau_F^1\ \ldots\ \tau_F^{n-1})$$

Indexed Predicate Term[-]:

$$\tau_{P,\pi} \coloneqq\ [\mathtt{p}\ (\tau_F^0\ \tau_F^1\ \ldots\ \tau_F^{n-1})]_\pi$$

Indexed Update[-]: $[\![\mathtt{s_o} \leftarrowtail \tau_F]\!]_\pi$

The restriction we imposed for updates now also holds for predicates, i.e., the trace variable indexing the predicate term determines the trace of the arguments. All arguments of the predicate come from this trace which is why we do not need to index

the arguments. In consequence, we do not need to index function terms either, as they can only occur as arguments of a predicate or in an update and, thus, their trace is determined. As $\mathcal{T}_P \subseteq \mathcal{T}_F$, an indexed predicate term can occur in an update but must be indexed with the same trace variable as the update. Predicates can also be nested, where again the outermost trace variable determines the index of the predicate terms occurring as arguments. As they must be from the same trace, we can omit the index when predicate terms occur as arguments and just use standard predicate terms.

While a HyperTSL⁻ formula is constructed from the same grammar as a HyperTSL formula and has the same semantics, the case for predicate evaluation can be simplified as we know that the arguments all come from the same trace. Using the same semantics rule as for HyperTSL, we would use the evaluation function $\mu_{\langle\cdot\rangle}$ to evaluate the predicate. We would then treat a predicate of the form $[\ \mathtt{p}\ (\tau_0\ \ldots\ \tau_{n-1})\ ]_\pi$ as if it was $\mathtt{p}\ (\tau_{0,\pi}\ \ldots\ \tau_{n-1,\pi})$ where each argument was indexed with the same trace variable π. This case can be simplified, as we have the additional information, that all arguments of the predicate come from the same trace. Therefore, we do not need the entire trace assignment at hand, when evaluating the components of the predicate. The concrete trace π suffices. Therefore instead of using $\mu_{\langle\cdot\rangle}$ we can use $\eta_{\langle\cdot\rangle}$ (see Definition 2.8) and provide it with the computation and input stream of trace π. The computation and the input stream are obtained by looking up the trace variable π in Π and then projecting on the first and second component respectively. Apart from predicate evaluation, a HyperTSL⁻ formula is evaluated using the same semantics as for HyperTSL as provided in Section 3.4. We define the satisfaction of predicates in a HyperTSL⁻ formula with respect to a trace set T^-, trace assignment $\Pi^- : \mathcal{V} \to (\mathcal{C}^\omega \times \mathcal{I}^\omega)$, and a time point $t \in \mathbb{N}$ as follows:

Definition 3.6 (Semantics HyperTSL⁻)

$$\Pi^-, T, t \models_{\langle\cdot\rangle} [\ \mathtt{p}\ (\tau_0\ \ldots\ \tau_{n-1})\ ]_\pi \Leftrightarrow\ \eta_{\langle\cdot\rangle}\ (\#1(\Pi^-(\pi)), \#2(\Pi^-(\pi)), t, \mathtt{p}\ (\tau_0\ \ldots\ \tau_{n-1}))$$

We use the evaluation function $\eta_{\langle\cdot\rangle}$ and provide it with the computation and input stream that form the trace we obtain when looking up π in the trace assignment Π^-. As we see, predicate evaluation in the HyperTSL⁻ fragment is contained in the semantics of the full hyperlogic. Nevertheless, for readability reasons and clarity we use the semantics as defined in Definition 3.6 for HyperTSL⁻ predicate terms. This simplified semantics is especially helpful in proofs to reduce notation overhead as we see in Chapter 5. While traces for HyperTSL and HyperTSL⁻ are of the same type, we annotate HyperTSL⁻ trace sets and assignments with a minus, in order to make it clear whether we are reasoning about the HyperTSL⁻ fragment or the full logic.

The hyperproperties we expressed in Section 3.2 so far all contain predicates that take arguments from different traces. However, for many important hyperproperties, it suffices to only evaluate predicates over arguments of one trace. For instance, we can

state the hyperproperty that if on both traces the `sun` values are always equal to the threshold `t` and the two traces consume energy in the same way, then the `bat` values on both traces are always decremented at the same time.

$$\begin{aligned}\forall\pi\forall\pi'.(\square\ ([\texttt{equal}(\texttt{sun},\texttt{t})]_\pi\ \wedge\ [\texttt{equal}(\texttt{sun},\texttt{t})]_{\pi'} \\ \wedge\ [\texttt{p consume}]_\pi \leftrightarrow [\texttt{p consume}]_{\pi'})) \\ \rightarrow\square\ ([\![\texttt{bat} \leftarrow\!\!\prec \texttt{dec bat}]\!]_\pi \leftrightarrow [\![\texttt{bat} \leftarrow\!\!\prec \texttt{dec bat}]\!]_{\pi'})\end{aligned} \tag{3.14}$$

Furthermore, we can state that if on both traces the battery is constantly charging and the energy consumption is the same, then all incrementations of battery values happen at the same time.

$$\begin{aligned}\forall\pi\forall\pi'.(\square\ ([\texttt{greater}(\texttt{sun},\texttt{t})]_\pi\ \wedge\ [\texttt{greater}(\texttt{sun},\texttt{t})]_{\pi'} \\ \wedge\ [\texttt{p consume}]_\pi \leftrightarrow [\texttt{p consume}]_{\pi'})) \\ \rightarrow\square\ ([\![\texttt{bat} \leftarrow\!\!\prec \texttt{inc bat}]\!]_\pi \leftrightarrow [\![\texttt{bat} \leftarrow\!\!\prec \texttt{inc bat}]\!]_{\pi'})\end{aligned} \tag{3.15}$$

Intuitively, both hyperproperties express that if the inputs behave similarly with regards to certain aspects, then the control flow behavior will be the same. Both hyperproperties hold on the system we specified.

HyperTSL and HyperTSL$^-$ allow expressing hyperproperties involving infinite data that were not expressible using HyperLTL. Furthermore, both allow a very intuitive description of the control flow in software, as thanks to predicates and updates the properties resemble control flow statements and assignments like in software code. HyperTSL allows expressing many powerful hyperproperties as predicates can take arguments from different traces. Thus it is, for instance, very useful for comparing values from different traces and specifying hyperproperties that describe the desired behavior based on whether the values were less, greater or equal to each other. Since functions and predicates are uninterpreted, we can specify a hyperproperty that describes a general behavior, which can be refined by choosing concrete function and predicate implementations and, thus, results in several hyperproperties specifying different instances of a general behavior. Even though the HyperTSL$^-$ fragment does not allow predicates with arguments from different traces, it can express many useful hyperproperties. It can use predicates to specify similar behavior of traces, for instance, that on both traces a certain predicate holds in the same time points. Again, by giving the predicates a different semantics, many instances of the same hyperproperty can be derived. While for HyperTSL we can directly compare values and for instance check for equality using an `equal` predicate, this is not possible for HyperTSL$^-$. If the same effect as for an equality predicate with arguments from different traces is to be achieved with HyperTSL$^-$, this needs to be encoded by an equality predicate that in each time step compares some fixed value `x` with the values `value` that should

be checked for equality on each trace respectively. If the values are from a finite domain with n elements, this can be encoded as a sequence of n conjuncts of the form $\bigwedge_{\mathtt{x}=0}^{n}([\mathtt{equal}\ (\mathtt{value}, \mathtt{x})]_{\pi} \leftrightarrow [\mathtt{equal}\ (\mathtt{value}, \mathtt{x})]_{\pi'})$. If the values are not from a finite domain, an equality predicate for values from different traces cannot be encoded in HyperTSL⁻. This shows that HyperTSL⁻ is less expressive than HyperTSL when it comes to directly comparing values with possibly infinite domains from different traces.

Chapter 4

Use Case: Diesel Scandal

In 2015, Diesel cars produced by the Volkswagen company were discovered to produce emissions high above the allowed values. Yet all the cars had been admitted to the road as they produced acceptable emissions in prior testing [19]. The software used in the exhaust control systems of the cars was manipulated to produce low emissions in testing scenarios, while producing 4 to 7 times higher emissions when on the road [14]. Worldwide, 11 million cars were affected of which 2.8 million in Germany alone [18]. The estimated cost of emission-related health issues and premature deaths entailed by the higher than standard emissions in the US and Europe in the years 2009 to 2015 is at 39 billion US Dollars [15]. The Diesel scandal is only one example of software doping. Software doping is a highly relevant topic that D'Argenio et al. have formalized in [4]. They define robust cleanness, or "doping-freedom" of a program, which intuitively means that two executions with similar inputs should produce similar outputs. Whether two inputs are "similar" is defined based on the distance that they are apart. Based on the allowed distance for similar inputs, the allowed distance for similar outputs is calculated. D'Argenio et al. state robust cleanness as a hyperproperty using HyperLTL and discretize the distance by using predefined values [4]. Hyperproperties expressing robust cleanness can be used for spotting doped software. In general, software doping describes the deliberate manipulation of software by the manufacturer in order to make it perform worse in certain scenarios such that it works in the best interest of the company but not anymore in the best interest of the user or society [4]. Other examples of software doping can be found in lock-in strategies that bind a customer to a company by building devices in ways such that additional supplies only work if they are from the same company. For instance, certain laptops refuse to charge when attached to a technically compatible charger from a different brand. Furthermore, some printers only work with cartridges from the same company, even though they are technically compatible with cartridges from other companies as well. Sometimes software in printers is also manipulated such that printers demand cartridge replacement even though the cartridge is not yet completely empty [4]. This way, more cartridges must be bought than would be technically needed

and the company profits. The Diesel scandal is a form of software doping that got a lot of attention in and after 2015 as it affected many people and the environment. It is especially shocking that it took such a long time for the manipulation to be discovered. A way of ensuring that emission regulation software in cars performs as desired at all times, and not only when attached to testing stations is needed. One way of spotting software doping before use is to use logic to specify the behavior of a doping-free system, as done by D'Argenio et al. [4]. Here, "doping-free" means that on similar inputs the system produces similar outputs in *all* cases, and not only when, for instance, attached to testing stations. In other words, the system is robustly clean.

In this chapter, we use HyperTSL to express hyperproperties that allow spotting doped software at the example of an emission regulation system, as the ones manipulated in the Diesel scandal. HyperTSL can express hyperproperties involving infinite data and can, thus, express instances of doping-freedom that cannot be expressed with HyperLTL. Furthermore, as HyperTSL uses uninterpreted functions and predicates, it allows expressing general hyperproperties that can be specialized for specific car models by using different function and predicate implementations.
In the following section, we gather some background knowledge on the test that cars have to pass in order to be admitted to the road, which is used to classify emission production as allowed or unallowed. Then, in Section 4.2, we specify properties of an emission regulation system using TSL. In Section 4.3, we use HyperTSL to specify instances of robust cleanness and other useful hyperproperties to spot software doping on the system that fulfills the properties as defined in Section 4.2.

4.1 The NEDC

In order to ensure that all exhaust regulation systems used in cars have a similar standard and produce emissions as low as possible, there is a test cycle that cars must take before being classified as allowed. This test cycle is the *New European Driving Cycle* (`NEDC`) [1, 5] which aims at representing the typical usage of a car in Europe. It is designed to help assess the emission levels of car engines and classify them as allowed or unallowed. The cycle is usually run while the car is attached to a specific test bench in order to eliminate external factors of inaccuracy such as weather conditions and road friction and to make the test cycle easily reproducible [1, 5, 12]. The cycle is composed of two main parts: the Urban Driving Cycle (`UDC`), and the Extra Urban Driving Cycle (`EUDC`) [5, 12]. The `UDC` is used to model the typical driving conditions that cars face in a busy city: frequent and abrupt starts and stops and a maximum speed of 50 km/h. In order to model driving conditions outside of the city, the `EUDC` represents the conditions for higher-speed driving. For the `EUDC`, the maximum speed is limited to 120 km/h [5, 12]. For both parts of the cycle, it is explicitly specified at what speed the car must drive for how long and in what gear. The `NEDC` was criticized for not being able to represent real-life driving conditions, due to these strict instructions [16]. In reality, hardly anyone can exactly time when

to shift gear and at what exact moment to break. Furthermore, there might also be other influential factors such as the driving style of the vehicle operator. It is, for instance, acceptable to use both the second or the third gear when driving 30 km/h. Cars should not only perform well when attached to a test bench or driving the exact values of the `NEDC` cycle, which we denote as `NEDCvalue` in the following, but also under standard driving conditions deviating from those checked in the test scenarios. Executions with driving values similar to those of the `NEDC` cycle should result in emissions similar to those described by the cycle. As we see, this property is an instance of robust cleanness. The Diesel exhaust emission scandal shed a light on the problem of specifying behavior on exhaust controlling software, which in the Diesel case was tailored to fit certain conditions. The cars passed the `NEDC` test, as the software was doped and designed to perform well on the test bench, yet when slightly deviating from the values used in the cycle emissions would heavily increase [19]. This type of manipulation falls under the previously mentioned notion of software doping. It is a negative property that cannot be spotted when only considering one execution of a test drive. One can only notice it when comparing several executions to each other and checking whether executions with similar speed and driving behavior produce similar emissions. This is because speed and driving behavior, such as the gear that is used for a specific speed and inclination, directly influence emission production.

4.2 Specifying the System

In this section, we specify properties that describe the behavior of an exhaust emission system. They can be checked either in real-time against the behavior of the car's emission control system or against the log data of the car's emission system produced after a test drive. The exhaust emission system we describe models the emission production of a vehicle in each time step based on the speed that the car is driving, as well as the gear and road inclination. The gear and inclination need to be considered as it can make a large difference in emission production what gear is being used and whether someone is driving up a hill or on a plain road. The necessary information to compute the emissions for each time step is obtained through the inputs `speed`, `gear`, and `inclination`. We define the `context` of an execution as the triple containing the values provided by `speed`, `gear`, and `inclination`. Storing these values allows re-accessing them in the next time step and is needed to evaluate the produced emissions. As we want the system to provide us with the amount of produced emissions for each time step, the computed value is available through the output cell `emissions`. Since the system aims at ensuring that emissions are sufficiently low, we want to be notified if emissions are critically high. Using the output cell `critical`, we constantly know whether emissions are too high. Furthermore, the system stores relevant information such as the runtime, the distance that has already been covered, and the average speed using a cell for each parameter.
While we are interested in the emissions produced per time step, it is also useful to

know the average emission value. This allows specifying hyperproperties where the average emissions are compared under the aspect of the driving style. For instance, it can be the case that two runs have the same average, while one of them has peaks and low points in emissions and the other has constant emissions. High speeds and coldstarts of cars often cause a temporary increase in emission [1], and a test run showing frequent starts and sections of high speed is more likely to have higher average emissions than a run that covers the same distance but with few stops and starts and mostly constant speed. As we also want to represent these two factors, we use the cell `maxSpeed` to store the maximum speed of the run so far and the cell `coldstart` that keeps track of how long it has been since the last start.
The cell `coldstart` is used to account for the fact that it is allowed to produce slightly higher emissions for a short time after a coldstart, as naturally, the car needs more power to get from standing still to moving than if it was just increasing the speed a little while already in motion. This cell is especially useful for modeling urban driving behavior. We first define the set of inputs $\mathbb{I}$, the set of outputs $\mathbb{O}$, and the set of cells $\mathbb{C}$, as well as the set of predicates $\mathcal{T}_P$ and the set of functions $\mathcal{T}_F$.

$$\begin{aligned}
\mathbb{I} &= \{\texttt{speed},\ \texttt{gear},\ \texttt{inclination}\} \\
\mathbb{O} &= \{\texttt{emissions},\ \texttt{critical}\} \\
\mathbb{C} &= \{\texttt{emissions},\ \texttt{critical},\ \texttt{context},\ \texttt{maxSpeed}, \texttt{coldstart}, \\
&\qquad \texttt{runtime},\ \texttt{distance},\ \texttt{avg_speed},\ \texttt{avg_emission}\} \\
\mathcal{T}_P &= \{\texttt{too_high},\ \texttt{greater},\ \texttt{equal}\} \\
\mathcal{T}_F &= \mathcal{T}_P \cup \{\texttt{compEm},\ \texttt{compDist},\ \texttt{compAS},\ \texttt{compAE},\ \texttt{inc}\}
\end{aligned}$$

We use uninterpreted functions and predicates when describing system properties and hyperproperties. However, in the following, we give them a semantics in order to make the properties and hyperproperties understandable. The function `compEm`(`speed`, `gear`, `inclination`) calculates the `emissions` that are produced with regard to the current `speed` and other factors such as the `gear` and the road `inclination`. The function `compDist`(`distance`, `speed`) calculates the distance that is already covered based on the `distance` that has already been covered until the previous time point and the `speed`. The function `compAS`(`distance`, `runtime`) computes the average speed of the execution using the already covered `distance` and the time that has passed, stored in `runtime`. In each time step, the average emissions are computed by the function `compAE`(`avg_emission`, `emissions`, `runtime`) based on the previous average `avg_emission`, the current `emissions`, and the `runtime`. The predicate `greater`(`x`, `y`) checks if the first argument `x` is greater than the second argument `y`. The predicate `equal`(`x`, `y`) checks for equality of the two arguments. In order to check whether the `emissions` that are produced are justified given the `context` and the time `coldstart` that it has been since the last start, we use the predicate `too_high`(`emissions`, `context`, `coldstart`). If emissions are too high, it evaluates to true.

We can now specify the desired behavior of an exhaust control system using TSL. We want the cell `context` to always be updated with a triple containing the values from the inputs `speed`, `gear`, and `inclination`. Storing these values allows re-accessing them in the next time step and is necessary for several other properties we want to specify.

$$\square \, [\![\texttt{context} \leftarrowtail (\texttt{speed}, \texttt{gear}, \texttt{inclination})]\!] \tag{4.1}$$

As we want to know the emissions that are produced in each step, we update the cell `emissions` with the amount of emissions that is produced based on `speed`, `gear` and `inclination` in each time step. Emissions are computed using the function `compEm`.

$$\square \, [\![\texttt{emissions} \leftarrowtail \texttt{compEm}(\texttt{speed}, \texttt{gear}, \texttt{inclination})]\!] \tag{4.2}$$

Note that we use the inputs directly to compute the emissions instead of using the triple stored in `context`. Using `context` would give us the context values from the previous time step, because the update takes one time step. We use the direct inputs in order to always have the `context` match the `emissions`. This means in each time step, the triple that is stored in `context` contains the values that caused the emissions stored in `emissions`. We could use what is stored in `context` to compute the emissions and assign `compEm` (`context`) to `emissions`, but then we would need an additional cell to store the context values from one time step before, as the cell `emissions` and `context` would always have an offset of one. We soon express a property for which we need both the `emissions` value and the `context` value from the same time step. Next, we specify a formula that increments the runtime in each step. This allows keeping track of how long the execution has taken.

$$\square \, [\![\texttt{runtime} \leftarrowtail \texttt{inc runtime}]\!] \tag{4.3}$$

We compute the distance that has already been covered in the execution using the function `compDist` which takes the distance that was calculated in the previous step and the speed we get as input.

$$\square \, [\![\texttt{distance} \leftarrowtail \texttt{compDist}(\texttt{distance}, \texttt{speed})]\!] \tag{4.4}$$

The cell `coldstart` is used to keep track of how long it has been since the last coldstart. If it is the case that the speed was at 0 km/h in one time step, meaning the car was standing still, and unequal to zero in the next one, then this means that the car just started again and it has been zero time steps since the last start.

$$\square \, (\texttt{equal}(\texttt{speed}, 0) \wedge \bigcirc \, \neg \texttt{equal}(\texttt{speed}, 0) \leftrightarrow \bigcirc \, [\![\texttt{coldstart} \leftarrowtail 0]\!]) \tag{4.5}$$

If and only if there was not a new coldstart, `coldstart` is incremented by one.

$$\square \, (\neg(\texttt{equal}(\texttt{speed}, 0) \wedge \bigcirc \neg \texttt{equal}(\texttt{speed}, 0)) \leftrightarrow \bigcirc [\![\texttt{coldstart} \leftarrowtail \texttt{inc coldstart}]\!]) \tag{4.6}$$

We update the cell `critical` with `true` whenever the emissions produced are higher than allowed. To determine if the emissions are higher than allowed we use the predicate `too_high`, that compares the actual emissions with what would be appropriate given the context.

$$\square\,(\texttt{too\_high}(\texttt{emissions}, \texttt{context}, \texttt{coldstart}) \leftrightarrow [\![\texttt{critical} \leftarrowtail \texttt{true}]\!]) \tag{4.7}$$

Note that `too_high` needs the emissions, but also the context that they were computed from in order to determine whether the emissions are appropriate given the context. This is why in Formula 4.2, we compute the emissions from (`speed`, `gear`, `inclination`) instead of `context` such that the values that are stored in `emissions` and `context` always match.
Even though there is an equivalence in the formula, we must explicitly cover the case where `too_high` does not hold. This is because without the case distinction it would be allowed to update the cell with its current value, $[\![\texttt{critical} \leftarrowtail \texttt{critical}]\!]$, if `too_high` does not hold. But if in the previous step, the cell was updated with `true`, then it contains the value `true` in the next time step again, even though `too_high` did not hold. Therefore, we need a property stating that if and only if `too_high` does not hold, we update the `critical` cell with `false`. As this distinction covers all cases, there is no more possibility for updates of the form $[\![\texttt{critical} \leftarrowtail \texttt{critical}]\!]$ to occur.

$$\square\,(\neg\texttt{too\_high}(\texttt{emissions}, \texttt{context}, \texttt{coldstart}) \leftrightarrow [\![\texttt{critical} \leftarrowtail \texttt{false}]\!]) \tag{4.8}$$

We compute the average speed of the execution using `distance` and `runtime` and store it in cell `avg_speed`.

$$\square\,[\![\texttt{avg\_speed} \leftarrowtail \texttt{compAS}(\texttt{distance}, \texttt{runtime})]\!] \tag{4.9}$$

The cell `avg_emission` is updated with the average emissions produced in the execution, computed by using the function `compAE` which needs the average amount of emissions computed so far, as well as the emissions of the current time step and the runtime.

$$\square\,[\![\texttt{avg\_emission} \leftarrowtail \texttt{compAE}(\texttt{avg\_emission}, \texttt{emissions}, \texttt{runtime})]\!] \tag{4.10}$$

As seen in the last two properties, the way we compute the average speed differs from the way we compute the average emissions. The average emissions could be computed from the sum of all emissions produced so far and the runtime. But then, we would need an extra cell to store the sum of all momentary emissions. We use an approach where we can compute the average emissions even without an extra cell using the average computed so far, the current emissions, and the runtime. Lastly, we want to store the maximum speed driven so far. Therefore, the cell `maxSpeed` is updated with the `speed` value if and only if it is the case that the value of `speed` is greater than the maximum speed stored so far.

$$\square\,(\texttt{greater}(\texttt{speed}, \texttt{maxSpeed}) \leftrightarrow [\![\texttt{maxSpeed} \leftarrowtail \texttt{speed}]\!]) \tag{4.11}$$

Here again, to be exhaustive, we must cover the case where the current speed is not greater than the maximum speed so far.

$$\square\,(\neg\texttt{greater}(\texttt{speed},\texttt{maxSpeed}) \leftrightarrow [\![\texttt{maxSpeed} \leftarrowtail \texttt{maxSpeed}]\!]) \tag{4.12}$$

In the next section, we specify hyperproperties to spot software-doping.

4.3 Specifying Hyperproperties

Naturally, different types of driving behavior cause different levels of emission production. However, there are certain general properties that a system should meet in order to be doping-free. In their paper, D'Argenio et al. specify what it means for software to be doping-free [4]. They introduce the notion of *robust cleanness*, which intuitively describes the property that small deviations in inputs should not result in huge deviations in outputs.
In the Diesel scandal, driving conditions that only slightly deviated from the `NEDC` values would cause emissions that heavily deviated from those produced under exact testing conditions. In reality, most likely no one drives under the exact conditions described by the cycle. For doped software, this may entail higher than allowed emissions though the software was declared conforming. Robust cleanness allows accounting for the fact that inputs that have a reasonably small distance should result in outputs with a reasonably small distance to each other. D'Argenio et al. introduce two notions of distance. The first one specifies the distance of two inputs, which is classified as acceptable based on a reference parameter stating the allowed deviation from the norm. The second notion specifies the allowed distance of two outputs depending on how much the inputs deviated. A program S is *robustly clean* if for all pairs of inputs, if the two inputs differ less than the allowed deviation from the norm, then the outputs produced by S under the respective inputs have a difference within an allowed range to each other [4]. D'Argenio et al. express robust cleanness as a HyperLTL formula. When using HyperLTL however, all the values from the `NEDC` that are relevant for the instance of robust cleanness that should be expressed must be discretized. This becomes difficult once data from an infinite domain is involved. Furthermore, what is considered a "small distance" must be fixed as a concrete value in advance whereas when using HyperTSL, uninterpreted predicates allow altering what is defined as a "small distance" after specifying the hyperproperty. In the following, we express instances of robust cleanness using HyperTSL. While we are interested in the behavior of traces where inputs are within an allowed deviation from the values defined by the `NEDC` to spot manipulation like in the Diesel scandal, we are also interested in the behavior of traces where the input parameters have a small distance to each other even though neither of them might be within the allowed distance from the `NEDC` values. Therefore, we specify hyperproperties that aim at spotting doped software by considering the exact `NEDC` values and slight deviations thereof, as well as hyperproperties where we compare values to each other without knowing their relation

to the NEDC values. Unlike for HyperLTL, we can use uninterpreted functions and predicates and, thus, do not need to use predefined values that specify the allowed distances for instance. Instead, we can choose function and predicate implementations afterward and alter their restrictiveness at will.

We introduce the following uninterpreted functions and predicates and again provide a semantics to make the hyperproperties we want to express more understandable. The function $\texttt{devNEDC}(\texttt{x})$ calculates how much the given value x deviates from the respective value of the NEDC cycle. It is thus defined as $\texttt{devNEDC}(\texttt{x}) = \texttt{NEDCvalue} - \texttt{x}$. This function is available for all parameters of the cycle. We make the functions distinct by indexing the function name with an abbreviation of the name of the value it checks. The function $\texttt{contextDev}(\texttt{s}, \texttt{g}, \texttt{i}) = (\texttt{devNEDC}_\texttt{S}\ (\texttt{s}), \texttt{devNEDC}_\texttt{G}\ (\texttt{g}), \texttt{devNEDC}_\texttt{I}\ (\texttt{i}))$ is used to check how much the speed s, the gear g and the inclination i deviate from the respective NEDC values. It returns a triple where in each position we have the deviation of the component from the respective NEDC values. The predicate $\texttt{smallDist}(\texttt{x}, \texttt{y})$ checks if the two arguments x and y only have a small distance to each other. The predicate $\texttt{smallDist}(\texttt{x}, \texttt{y})$ is available for all parameters of the cycle. We make the predicates distinct by indexing their names with an abbreviation of the name of the value they check. Whether a distance is small or not can be for instance defined depending on the model of the car or other specific parameters. For which values the predicate evaluates to true can thus be modified for each car model as desired. This is a major advantage of HyperTSL: uninterpreted functions and predicates allow specifying hyperproperties with some general structure, that can later be refined depending on how the concrete implementation for functions and predicates is chosen. In the case of software doping, this allows specifying one hyperproperty that can be used for different car models, for instance, by changing the predicate implementation that classifies what is to be considered a small distance when comparing gears.

The following hyperproperties help to spot undesired behavior in a car's exhaust emission controller. We first express some hyperproperties using HyperTSL and afterward have a look at hyperproperties expressed in HyperTSL$^-$. The first hyperproperty describes an instance of robust cleanness. The cell context contains all the relevant input information that influences the emissions. For all pairs of traces, it must always hold that if the contexts are similar, meaning their pairwise respective values only have a small distance, then the emissions should also only have a relatively small distance.

$$\begin{aligned} \forall\pi\forall\pi'.\square\ (&\texttt{smallDist}_\texttt{C}(\texttt{context}_\pi, \texttt{context}_{\pi'}) \\ &\rightarrow \texttt{smallDist}_\texttt{E}(\texttt{emissions}_\pi, \texttt{emissions}_{\pi'})) \end{aligned} \tag{4.13}$$

An example is given by the contexts (30 km/h, 2, 7°) and (30 km/h, 3, 7°). The emissions that are produced for these two contexts should only differ within a reasonably small range. What "reasonably" small means can be defined individually depending on how restricted the system should be in its behavior.

The next hyperproperty expresses something similar to Formula 4.13, but this time with respect to the NEDC values. It is also an instance of robust cleanness. It must always be the case that if the two traces have a similar but still acceptably small deviation from the values provided by the test cycle, the emissions also must have a similar, still acceptable deviation from the NEDC emission values.

$$\begin{aligned}\forall\pi\forall\pi'.\square\ (&\texttt{smallDist}_{\texttt{C}}(\texttt{contextDev}(\texttt{context}_{\pi}), \texttt{contextDev}(\texttt{context}_{\pi'})) \\ &\rightarrow \texttt{smallDist}_{\texttt{E}}(\texttt{devNEDC}_{\texttt{E}}(\texttt{emissions}_{\pi}), \texttt{devNEDC}_{\texttt{E}}(\texttt{emissions}_{\pi'})))\end{aligned} \tag{4.14}$$

As an example, consider two traces where the contexts always deviate from the NEDC contexts only in a speed difference of 5 km/h. The other context parameters are the same. The trace π has a speed of $\texttt{NEDC}_{\texttt{speed}} + 5$ km/h and trace π' a speed of $\texttt{NEDC}_{\texttt{speed}} - 5$ km/h. Assume the $\texttt{smallDist}_{\texttt{C}}$ predicate is fulfilled. The emissions should now behave accordingly. How much they are allowed to deviate from the expected values and how large the deviations are allowed to be with respect to each other can depend on the car model or on how strict one wants the emissions to be limited.

The next hyperproperty accounts for the fact that after a coldstart, it is allowed to produce more emissions than usual for some predefined time. Initially, the contexts must be similar, and on one of the two traces, there is a coldstart, while for the predefined number of time steps there is no new coldstart on either trace and the contexts are similar. After this predefined time, emission production must go back to normal if the contexts are similar again. This allows for higher emission production after a coldstart, but does not enforce it. We specify the hyperproperty using 5 as the predefined time.

$$\begin{aligned}\forall\pi\forall\pi'.\square\ \Bigg(&\bigwedge_{i=0}^{5} \bigcirc_i\ \texttt{smallDist}_{\texttt{C}}(\texttt{context}_{\pi}, \texttt{context}_{\pi'}) \\ &\wedge\ (\llbracket \texttt{coldstart} \leftarrow 0 \rrbracket_{\pi} \oplus \llbracket \texttt{coldstart} \leftarrow 0 \rrbracket_{\pi'}) \\ &\wedge\ \bigwedge_{i=1}^{5} \bigcirc_i\ (\neg\llbracket \texttt{coldstart} \leftarrow 0 \rrbracket_{\pi} \wedge \neg\llbracket \texttt{coldstart} \leftarrow 0 \rrbracket_{\pi'}) \\ &\rightarrow\ \bigcirc_5\ (\texttt{smallDist}_{\texttt{E}}(\texttt{emissions}_{\pi}, \texttt{emissions}_{\pi'}))\Bigg)\end{aligned} \tag{4.15}$$

This hyperproperty can be adapted to be more strict by changing the predefined number of time steps. For instance, by choosing 0 as the predefined time where higher emission production would be accepted, we can express that coldstarts should not produce higher emissions than standard driving behavior. This is a property that might, for instance, hold on cars that are designed to be especially efficient when starting.
The next hyperproperty states that it should always be the case that if both the

average emissions and average speeds are similar, then it must also be that the two traces have similar maximum speed values and that the last start has been at about the same time ago. The hyperproperty is based on the assumption that driving at a significantly higher speed and stopping and starting more frequently have a large impact on the emission production.

$$\begin{aligned} \forall\pi\forall\pi'.\quad \square\,(&\texttt{smallDist}_{\texttt{AS}}(\texttt{avg\_speed}_{\pi}, \texttt{avg\_speed}_{\pi'}) \\ \wedge\ &\texttt{smallDist}_{\texttt{AE}}(\texttt{avg\_emission}_{\pi}, \texttt{avg\_emission}_{\pi'}) \\ \rightarrow\ &\texttt{smallDist}_{\texttt{MS}}(\texttt{maxSpeed}_{\pi}, \texttt{maxSpeed}_{\pi'}) \\ \wedge\ &\texttt{smallDist}_{\texttt{CS}}(\texttt{coldstart}_{\pi}, \texttt{coldstart}_{\pi'})) \end{aligned} \tag{4.16}$$

If the hyperproperty is not fulfilled, it is worth checking why the average emissions are roughly the same. If it is, for instance, the case that the average speed and average emissions are about the same, yet the maximum speeds or the time after the last coldstarts differ a lot, then in at least one trace there must be some source of higher emission production that might be an indication of software doping.

While the previous hyperproperties were all expressed using HyperTSL, we now express some hyperproperties using HyperTSL^{-}. We use the following uninterpreted functions and predicates which we give the following semantics for this example. The constant `dev` defines how much the inputs are allowed to deviate from the `NEDC` values and is chosen depending on how restrictive the system should be. The function `outdev(dev)` computes how much the outputs are allowed to deviate from the `NEDC` values depending on how much the inputs deviated. The predicate $\texttt{allowed}_{\texttt{C}}(\texttt{dev}, \texttt{c})$ determines if the context deviation `c` is within the allowed input deviation range. The predicate `allowedDev(y, devNEDC(x))` checks if the actual deviation of the value `x` from the `NEDC` values complies with the allowed distance `y`. It is available for all parameters of the cycle and depending on the value it checks uses the matching `devNEDC` predicate. The `goodStyle(context)` predicate, checks if the driving behavior represented by the context would be classified as a "good style". We define a good driving style for our example as follows: The speed matches the gear and inclination and aims at producing optimal emissions.

The first hyperproperty we express using HyperTSL^{-} is again an instance of robust cleanness. It states that for all pairs of traces, if contexts always only deviate within the predefined allowed range `dev` from the `NEDC` values on both traces in the same time steps, then the produced emissions should also always deviate within the allowed range `outdev(dev)` in the same time steps as well.

$$\begin{aligned} \forall\pi\forall\pi'.\ \square\quad &([\texttt{allowed}_{\texttt{C}}(\texttt{dev}, \texttt{contextDev (context)})]_{\pi} \\ \leftrightarrow\quad &[\texttt{allowed}_{\texttt{C}}(\texttt{dev}, \texttt{contextDev (context)})]_{\pi'}) \\ \rightarrow\ \square\quad &([\texttt{allowedDev}_{\texttt{E}}(\texttt{outdev(dev)}, \texttt{devNEDC}_{\texttt{E}}(\texttt{emissions}))]_{\pi} \\ \leftrightarrow\quad &[\texttt{allowedDev}_{\texttt{E}}(\texttt{outdev(dev)}, \texttt{devNEDC}_{\texttt{E}}(\texttt{emissions}))]_{\pi'}) \end{aligned} \tag{4.17}$$

The next hyperproperty uses the predicate `goodStyle` to decide whether the gear chosen for the given speed and inclination is appropriate and aims at a good driving style and at producing low emissions. Generally, it should be the case that exhaust control systems are designed to produce optimal emissions. An appropriate driving style can be beneficial for low emission production. Under this assumption, the hyperproperty states that if it is the case that on both traces the driving behavior is the same with respect to the style and the starts and stops happen at the same time, then the `critical` cells on both traces should also only be updated with `true` in exactly the same time steps.

$$\begin{aligned} \forall\pi\forall\pi'.\square\,(([\texttt{goodStyle}(\texttt{context})]_\pi \leftrightarrow [\texttt{goodStyle}(\texttt{context})]_{\pi'})a \\ \wedge\,([\![\texttt{coldstart} \longleftarrow 0]\!]_\pi \leftrightarrow [\![\texttt{coldstart} \longleftarrow 0]\!]_{\pi'})) \\ \rightarrow\square\,([\![\texttt{critical} \longleftarrow \texttt{true}]\!]_\pi \leftrightarrow [\![\texttt{critical} \longleftarrow \texttt{true}]\!]_{\pi'}) \end{aligned} \tag{4.18}$$

As we have seen, both HyperTSL$^-$ and HyperTSL can express many useful hyperproperties that specify doping-free behavior. When software reasons about infinite data, we cannot use HyperLTL to specify hyperproperties. Software in cars often relies on multiple sensors and the provided data is often from infinite domains. Therefore, when specifying behavior on this type of software, we need to use HyperTSL. One reason to use HyperTSL even for software with finite data is that functions and predicates are uninterpreted. Therefore, hyperproperties can be specified and later refined depending on the concrete function and predicate implementation that is chosen.

Chapter 5

Synthesis

In synthesis, a system is constructed from a specification. The resulting system is guaranteed to fulfill the properties defined by the specification by construction. While LTL synthesis is very useful, it has some deficits when it comes to large specifications involving huge amounts of data. Since LTL expresses data using atomic propositions, infinite domains are not expressible. To overcome this problem, Finkbeiner et al. introduce the logic TSL [10] which succeeds in expressing large specifications with infinite data. TSL abstracts from concrete data and uses predicates and functions to reason about the control flow in a system. Due to its higher expressivity, TSL can describe specifications reasoning about infinite data. Therefore, scenarios that could not be synthesized using LTL as specification language could be synthesized using TSL [10]. However, expressivity of TSL in comparison to LTL comes at the cost of decidability.

While synthesis from an LTL or TSL specification constructs a system that is guaranteed to fulfill the specification, there is no guarantee whether the system will also fulfill certain hyperproperties. Synthesis from hyperproperties, more specifically HyperLTL synthesis, is described by Finkbeiner et al. in [11]. While systems can be synthesized based on a specification expressed in LTL and HyperLTL, there is not yet a possibility to synthesize systems from a TSL specification and guarantee that they fulfill certain hyperproperties as well. Especially hyperproperties reasoning about infinite data, which cannot be specified using HyperLTL, can thus not be synthesized so far.

Therefore, in this chapter, we introduce synthesis for HyperTSL$^-$. Similar to TSL synthesis, which is based on an approximation to LTL, HyperTSL$^-$ synthesis is based on an approximation to HyperLTL. Synthesis for HyperTSL and synthesis for HyperTSL$^-$ are undecidable, as TSL synthesis is already undecidable.

We present a bounded synthesis approach for the universal HyperTSL$^-$ fragment of HyperTSL. We first explain, why we restrict ourselves to the HyperTSL$^-$ fragment of HyperTSL and later explain, why we restrict ourselves to the universal fragment of HyperTSL$^-$. Intuitively, finding an approach for the HyperTSL synthesis problem is more challenging than for the HyperTSL$^-$ synthesis problem. This is because for

HyperTSL$^-$ synthesis, we can encode components that contain information about one trace by atomic propositions in HyperLTL which also contain information about one trace, while for HyperTSL synthesis we would need to encode components that contain information about multiple traces by components in HyperLTL which can only contain information about *one* trace. For the HyperTSL$^-$ fragment, we find an approximation to HyperLTL as we explore later. We have a quick look at the approximation challenge for HyperTSL: In HyperTSL, predicates can take arguments from different traces. Therefore, the arguments of a predicate are indexed with several, potentially different trace variables. Finding a HyperLTL encoding for a predicate of this form becomes difficult. Predicates can no longer be seen as atomic components, and the approach of introducing *one* atomic proposition with a trace variable as index becomes impossible without completely changing the actual meaning of the predicate term. While this loss of semantics can be reverted when approximating a formula of the HyperTSL$^-$ fragment, finding a straightforward way of restoring the meaning for a HyperTSL formula is rather difficult. Furthermore, bounded synthesis of the HyperTSL$^-$ fragment is especially attractive since already existing approaches can be adapted and used. For HyperTSL synthesis a new approach might be needed. Before we have a look at the general structure of the bounded synthesis process for HyperTSL$^-$, we define the necessary concepts for HyperTSL$^-$ synthesis and provide some intuition.

In synthesis, the aim is to construct a *strategy* that describes with what output a system answers to an input. The input-output combinations resulting from the strategy must fulfill the specification. Since for HyperTSL$^-$ the inputs are the predicates that hold and the outputs are the computation steps that result, a HyperTSL$^-$ strategy f has the type $f : (2^{\mathcal{T}_P})^+ \to \mathcal{C}$. It is represented by a strategy tree that branches on all inputs and that has nodes labeled with the output with which the system reacts. Figure 5.1 shows a strategy tree for one predicate p.

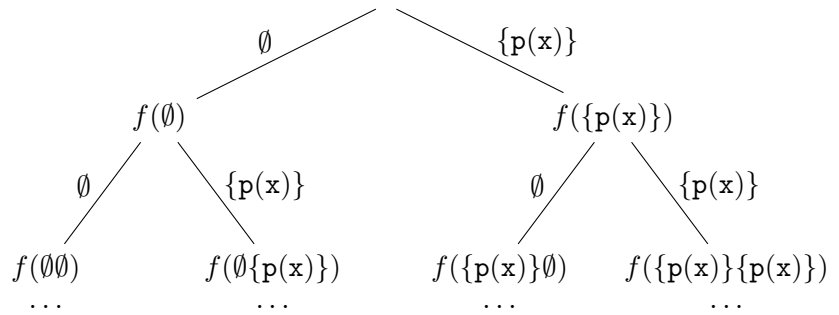

Figure 5.1: Strategy tree f of a HyperTSL$^-$ strategy $f : (2^{\mathcal{T}_P})^+ \to \mathcal{C}$

A strategy *realizes* a specification if the set of traces that can be collected from the strategy tree satisfies the specification. A HyperTSL$^-$ formula is *realizable* if a strategy

exists such that the set of traces resulting from the tree satisfies the formula. We formally define HyperTSL$^-$ realizability as follows:

Definition 5.1 (HyperTSL$^-$ Realizability)
A HyperTSL$^-$ formula φ is realizable if there exists a strategy $f : (2^{\mathcal{T}_P})^+ \to \mathcal{C}$, such that for every function assignment $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$, the set of traces obtained from all combinations of inputs and corresponding computations generated by the strategy satisfies the formula. Formally:

$$\exists f : (2^{\mathcal{T}_P})^+ \to \mathcal{C}.\ \forall \langle\cdot\rangle : \mathbb{F} \to \mathcal{F}.\ \{(\sigma, \iota) \mid \iota \in \mathcal{I}^\omega \wedge \forall t \in \mathbb{N}.$$
$$\sigma(t) = f(\{\tau_P \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(\sigma, \iota, 0, \tau_P)\} \ldots \{\tau_P \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(\sigma, \iota, t, \tau_P)\})\} \models_{\langle\cdot\rangle} \varphi$$

As we see in the definition, it is important that the strategy works for all function evaluations, as we work with uninterpreted functions and predicates. The traces that are contained in the set are all pairs of input-computation combinations such that for each input, the corresponding computation is constructed from the strategy. Each computation step $\sigma(t)$ is defined as the computation step returned by the strategy when given the sequence of the sets of all predicates that hold given the input stream and computation so far.

For the synthesis approach we describe, we only consider universal HyperTSL$^-$ formulas, as we overapproximate the HyperTSL$^-$ formula with an HyperLTL formula. We explain why this overapproximation requires us to restrict HyperLTL to the universal fragment when looking at the approximation in detail. We now first explain the general structure of bounded HyperTSL$^-$ synthesis,which follows the graphical representation in Figure 5.2, before having a detailed look at each step. First, we approximate the universal HyperTSL$^-$ formula using HyperLTL. To do so, we encode predicate terms and updates as atomic propositions. Naturally through this, the semantics of updates and predicates used in the HyperTSL$^-$ formula get lost and need to be restored at some point. The semantics of predicates is restored during the synthesis process by iteratively adding constraints to the formula. The semantics of updates is restored by adding a constraint to the HyperLTL encoding before the synthesis process starts. The approximated HyperLTL formula, thus, is a conjunction of the update restoration and the HyperLTL encoding and can be synthesized using standard HyperLTL tools for bounded synthesis, such as BoSyHyper. If the tool returns that the formula is realizable, we know that the HyperTSL$^-$ formula is also realizable. If the result is that the formula is not realizable, the tool provides a counter strategy. This counter strategy must be checked for spuriousness as it can be that this result was caused by an imprecise approximation and does not necessarily mean that the HyperTSL$^-$ formula is not realizable. It is rather an indication that the HyperLTL formula must be refined in order to precisely express the original predicate semantics. If the counter strategy is indeed spurious, we need to add a refinement to the formula and use the synthesis tool again. If, however, the check reveals that the counter strategy is not spurious, then the HyperTSL$^-$ formula is not realizable.

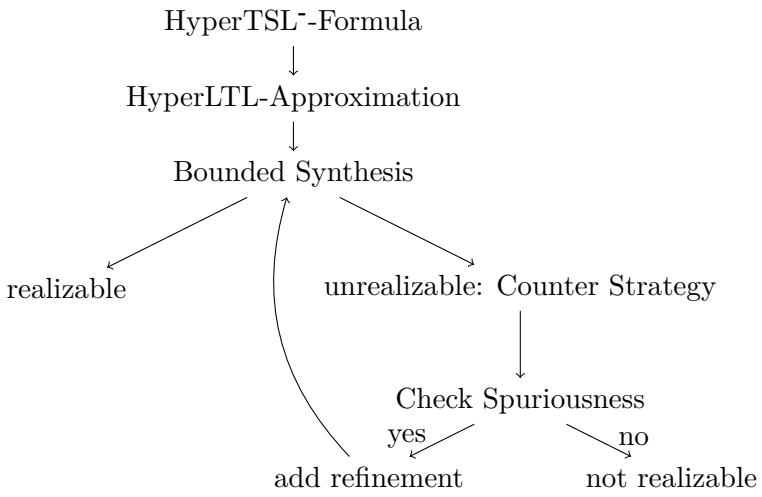

Figure 5.2: Structure of HyperTSL$^-$ Synthesis

We now have a more detailed look at the individual steps of the synthesis process. In order to approximate a HyperTSL$^-$ formula with a HyperLTL formula, we must transform the individual components. Quantifiers, as well as the initial structure of the formula, remain the same, but predicates and updates must be translated into a suitable HyperLTL encoding, i.e., into atomic propositions. We formally define the syntactic conversion.

Definition 5.2 (Syntactic Conversion)
We define the syntactic conversion of a HyperTSL$^-$-formula φ_- inductively as follows:

$$
\begin{aligned}
&\texttt{SynCon}([\texttt{p}\ \tau_{F_1}\ \ldots \tau_{F_n}]_\pi) = \texttt{p\_}\tau_{\texttt{F}_1}\texttt{\_}\tau_{\texttt{F}_2}\texttt{\_}\cdots\texttt{\_}\tau_{\texttt{F}_n\pi}\\
&\texttt{SynCon}([\![\texttt{s} \leftarrowtail \tau_F]\!]_\pi) = \texttt{s\_up\_}\tau_{\texttt{F}_\pi}\\
&\texttt{SynCon}(\forall\pi.\varphi) = \forall\pi.\texttt{SynCon}(\varphi)\\
&\texttt{SynCon}(\varphi_1 \wedge \varphi_2) = \texttt{SynCon}(\varphi_1) \wedge \texttt{SynCon}(\varphi_2)\\
&\texttt{SynCon}(\bigcirc\varphi) = \bigcirc\texttt{SynCon}(\varphi)\\
&\texttt{SynCon}(\neg\varphi) = \neg\texttt{SynCon}(\varphi)\\
&\texttt{SynCon}(\varphi_1\,\mathcal{U}\,\varphi_2) = \texttt{SynCon}(\varphi_1)\ \mathcal{U}\ \texttt{SynCon}(\varphi_2)
\end{aligned}
$$

SynCon of a set that contains predicate terms or updates is the set that results when applying SynCon to each element. SynCon of a sequence of sets is the sequence that results when applying SynCon to each set. We define $\widehat{S} = \texttt{SynCon}(S)$ to be the syntactic conversion of a set S.

An update $[\![\mathtt{s} \leftarrowtail \tau_F]\!]_\pi$ is encoded by a unique atomic proposition indexed with the same trace variable π. The same transformation is applied to indexed predicate terms, where each term is encoded by a unique atomic proposition indexed with the same trace variable. For formulas containing quantifiers or operators, the syntactic conversion is moved inwards until applied to a predicate or an update. The formula we obtain after these steps is denoted by $\mathtt{SynCon}(\varphi_\text{-})$.

Example 5.1 (Example Syntactic Conversion)
We have a look at an example formula $\varphi_\text{-}$ which also serves as our running example throughout this section. The formula states that if on a pair of traces, the cell `x` is always updated with `y` in exactly the same time steps on both traces, if the predicate `p y` also holds in exactly the same time steps on both traces, and if it eventually evaluates to true on one trace, then eventually `p x` holds on both traces.

$$\varphi = \forall\pi\forall\pi'.(\square\,([\![\mathtt{x} \leftarrowtail \mathtt{y}]\!]_\pi \leftrightarrow [\![\mathtt{x} \leftarrowtail \mathtt{y}]\!]_{\pi'}) \;\wedge\; \square\,([\mathtt{p\ y}]_\pi \leftrightarrow [\mathtt{p\ y}]_{\pi'}) \;\wedge\; \diamondsuit\,[\mathtt{p\ y}]_\pi) \rightarrow \diamondsuit\,([\mathtt{p\ x}]_\pi \;\wedge\; [\mathtt{p\ x}]_{\pi'})$$

The formula is realizable by updating `x` with `y` as soon as `p y` holds on trace π. The syntactic conversion of the formula looks as follows:

$$\forall\pi\forall\pi'.(\square\,(\mathtt{x\_up\_y}_\pi \leftrightarrow \mathtt{x\_up\_y}_{\pi'}) \;\wedge\; \square\,(\mathtt{p\_y}_\pi \leftrightarrow \mathtt{p\_y}_{\pi'}) \;\wedge\; \diamondsuit\;\mathtt{p\_y}_\pi) \rightarrow \diamondsuit\,(\mathtt{p\_x}_\pi \;\wedge\; \mathtt{p\_x}_{\pi'})$$

Naturally, through this transformation, we lose the semantics of predicates and updates. The semantics of predicates will be partially restored during the synthesis process by adding refinements to the formula. The semantics of updates is restored by an additional constraint that is added before the synthesis process starts. Let $\mathcal{T}_{P,\mathcal{V}}$ and $\mathcal{T}_{U,\mathcal{V}}$ be the finite sets of indexed predicate terms and updates respectively which appear in the HyperTSL$^-$ formula. Let Q be the set containing exactly the trace variables that occur in a HyperTSL$^-$ formula. For every indexed update that occurs in the formula, we partition $\mathcal{T}_{U,\mathcal{V}}$ into $\uplus_{\pi\in Q}\uplus_{\mathtt{s_o}\in\mathbb{C}\cup\mathbb{O}}\mathcal{T}^{\mathtt{s_o}}_{U,\pi}$ where $\uplus$ is the disjoint union of sets. Intuitively, $\mathcal{T}^{\mathtt{s_o}}_{U,\pi}$ denotes the set containing all updates to a cell $\mathtt{s_o}$ on trace π. For every cell $\mathtt{c}\in\mathbb{C}$ we define $\mathcal{T}^{\mathtt{c}}_{U,\pi/\mathtt{id}} = \mathcal{T}^{\mathtt{c}}_{U,\pi}\cup\{[\![\mathtt{c} \leftarrowtail \mathtt{c}]\!]\}$, for $\mathtt{o}\in\mathbb{O}$ we define $\mathcal{T}^{\mathtt{o}}_{U,\pi/\mathtt{id}} = \mathcal{T}^{\mathtt{o}}_{U,\pi}$ and $\mathcal{T}_{U,\pi/\mathtt{id}} = \bigcup_{\mathtt{s_o}\in\mathbb{C}\cup\mathbb{O}}\mathcal{T}^{\mathtt{s_o}}_{U,\pi/\mathtt{id}}$. Lastly, we define $\mathcal{T}_{U,Q/\mathtt{id}} = \bigcup_{\pi\in Q}\mathcal{T}_{U,\pi/\mathtt{id}}$. The meaning of updates is restored by the constraint that on each trace and for each cell in any given time point exactly one update for this cell holds. This constraint is necessary, because updates are encoded as atomic propositions that could technically be all true at the same time.

Definition 5.3 (Restoration of Update Semantics)
Let Q be the set of all trace variables that occur in the formula. We define the constraint for update restoration as follows:

$$\square\,(\bigwedge_{\pi\in Q}\bigwedge_{\mathtt{s_o}\in\mathbb{C}\cup\mathbb{O}}\bigvee_{\tau\in\mathcal{T}^{\mathtt{s_o}}_{U,\pi/\mathtt{id}}}(\tau \;\wedge\; \bigwedge_{\tau'\in\mathcal{T}^{\mathtt{s_o}}_{U,\pi/\mathtt{id}}\setminus\{\tau\}}\neg\,\tau'))$$

Example 5.2 (Example Update Restoration)
Restoring the update semantics of our formula φ yields:

$$\begin{aligned}&\square\,((\llbracket \mathtt{x} \leftarrowtail \mathtt{y} \rrbracket_\pi \;\wedge\; \neg\, \llbracket \mathtt{x} \leftarrowtail \mathtt{x} \rrbracket_\pi \;\vee\; \neg\, \llbracket \mathtt{x} \leftarrowtail \mathtt{y} \rrbracket_\pi \;\wedge\; \llbracket \mathtt{x} \leftarrowtail \mathtt{x} \rrbracket_\pi)\\ &\wedge\; (\llbracket \mathtt{x} \leftarrowtail \mathtt{y} \rrbracket_{\pi'} \;\wedge\; \neg\, \llbracket \mathtt{x} \leftarrowtail \mathtt{x} \rrbracket_{\pi'} \;\vee\; \neg\, \llbracket \mathtt{x} \leftarrowtail \mathtt{y} \rrbracket_{\pi'} \;\wedge\; \llbracket \mathtt{x} \leftarrowtail \mathtt{x} \rrbracket_{\pi'}))\end{aligned}$$

The final HyperLTL formula approximating the HyperTSL$^-$ formula is a conjunction of the syntactically converted formula and the syntactic conversion of the previously defined semantics restoration. The HyperLTL formula is constructed over the input propositions $\mathcal{T}_{P,\mathcal{V}}$ and output propositions $\mathcal{T}_{U,Q/\mathtt{id}}$ as follows.

Definition 5.4 (Approximation)

$$\varphi_{HLTL} = \mathtt{SynCon}(\varphi_{\text{-}}) \wedge\; \mathtt{SynCon}(\square\,(\bigwedge_{\pi \in Q} \bigwedge_{\mathtt{s_o} \in \mathbb{C} \cup \mathbb{O}} \bigvee_{\tau \in \mathcal{T}^{\mathtt{so}}_{U,\pi/\mathtt{id}}} (\tau \;\wedge \bigwedge_{\tau' \in \mathcal{T}^{\mathtt{so}}_{U,\pi/\mathtt{id}} \setminus \{\tau\}} \neg\, \tau')))$$

Example 5.3 (Example Approximation)
Combining the syntactic conversion of the formula and the syntactic conversion of the restoration of the update semantics we get:

$$\begin{aligned}\forall\pi\forall\pi'.(&\square\,(\mathtt{x\_up\_y}_\pi \leftrightarrow \mathtt{x\_up\_y}_{\pi'}) \;\wedge\; \square\,(\mathtt{p\_y}_\pi \leftrightarrow \mathtt{p\_y}_{\pi'}) \;\wedge\; \diamondsuit\; \mathtt{p\_y}_\pi)\\ &\rightarrow \diamondsuit\,(\mathtt{p\_x}_\pi \;\wedge\; \mathtt{p\_x}_{\pi'})\\ &\wedge\; \square\,((\mathtt{x\_up\_y}_\pi \;\wedge\; \neg\, \mathtt{x\_up\_x}_\pi \;\vee\; \neg\, \mathtt{x\_up\_y}_\pi \;\wedge\; \mathtt{x\_up\_x}_\pi)\\ &\wedge\; (\mathtt{x\_up\_y}_{\pi'} \;\wedge\; \neg\, \mathtt{x\_up\_x}_{\pi'} \;\vee\; \neg\, \mathtt{x\_up\_y}_{\pi'} \;\wedge\; \mathtt{x\_up\_x}_{\pi'}))\end{aligned}$$

As we see, the approximated HyperLTL formula overapproximates the HyperTSL$^-$ formula. The trace set resulting from the HyperTSL$^-$ strategy tree according to the realizability definition depends on input streams and on how predicates evaluate under these input streams. There are certain sequences of predicate evaluations in the strategy tree, that can never be in the resulting trace set because there is no input stream that could cause such predicate evaluations. Meanwhile, in the trace set of the HyperLTL strategy tree, however, there exist such sequences, as the atomic propositions are independent, and thus all combinations of atomic propositions can be obtained. An example would be to have the branch $\mathtt{p\_i}\ \neg\mathtt{p\_i}\ \mathtt{p\_i}\ \neg\mathtt{p\_i}\ldots$ in the HyperLTL tree, while in the HyperTSL$^-$ tree, there is no input sequence that can cause this alternating predicate evaluation, $\mathtt{p(i)}\ \neg\mathtt{p(i)}\ \mathtt{p(i)}\ \neg\mathtt{p(i)}\ldots$, because predicates always evaluate the same under the same arguments at all time points. Therefore, there are more traces in the trace set resulting from the HyperLTL tree than in the trace set resulting from the HyperTSL$^-$ strategy tree. The set of traces obtained from the HyperTSL$^-$ tree according to the realizability definition can thus be seen as a "subset" of the trace set from the HyperLTL tree. Therefore, we must

restrict ourselves to the universal fragment of HyperLTL, as otherwise, we might find a realizing strategy for the approximation, which is not a realizing strategy for the original HyperTSL$^-$ formula. The problem shows when considering a formula with existential quantification, like:

$$\exists\pi.\square\,((\texttt{p\_i} \rightarrow \bigcirc\ \neg\texttt{p\_i})\ \wedge\ (\neg\texttt{p\_i} \rightarrow \bigcirc\ \texttt{p\_i}))$$

The trace $\texttt{p\_i}\ \neg\texttt{p\_i}\ \texttt{p\_i}\ \neg\texttt{p\_i}\ldots$ and the trace $\neg\texttt{p\_i}\ \texttt{p\_i}\ \neg\texttt{p\_i}\ \texttt{p\_i}\ldots$ are the only traces fulfilling the formula and are both contained in the HyperLTL trace set. In the trace set collected from the HyperTSL$^-$ strategy tree according to the realizability definition, however, no such traces exist, as no input stream could cause such predicate evaluations. For universal quantification, there is no problem, as we pick an arbitrary HyperTSL$^-$ trace from the HyperTSL$^-$ trace set, and find out whether the property holds based on whether the corresponding HyperLTL trace fulfills the property. In this direction, from HyperTSL$^-$ to HyperLTL, however, we can be sure to always find a corresponding trace.

Now that the universal HyperTSL$^-$-formula is converted into a universal HyperLTL-formula, we can use the HyperLTL synthesis tool BoSyHyper, which works for universal HyperLTL with up to one quantifier alternation [3]. This tool returns a HyperLTL strategy if the formula is realizable, and a counter strategy in the form of an infinite tree if it is unrealizable. We have a more detailed look at both cases.

If the formula is realizable, the tool returns a HyperLTL strategy of the type $h : (2^{\widehat{\mathcal{T}}_P})^+ \rightarrow (2^{\widehat{\mathcal{T}}_{U/\texttt{id}}})$ where $\widehat{\mathcal{T}}_P$ and $\widehat{\mathcal{T}}_{U/\texttt{id}}$ are the finite sets of syntactically converted predicate terms and updates respectively. Since the HyperLTL formula that resulted from syntactically converting the HyperTSL$^-$-formula and adding constraints, is realizable, the original HyperTSL$^-$-formula is realizable as well, as we show in Proof 5.2. A strategy is realizable iff the set of traces collected from the strategy tree satisfies the formula. Therefore, in order to get an understanding of how such a strategy for HyperTSL$^-$ can be constructed from a HyperLTL strategy, we first specify what it means for HyperLTL and HyperTSL$^-$ traces and trace assignments to be corresponding.

Definition 5.5 (Corresponding Traces)
A HyperLTL trace tr over $AP = \widehat{\mathcal{T}}_P \cup \widehat{\mathcal{T}}_{U/\texttt{id}}$ that is the result of a syntactic conversion, and a HyperTSL$^-$ trace $ts = (z, \iota)$ are corresponding with respect to a function evaluation $\langle\cdot\rangle$, denoted by $tr \sim_{\langle\cdot\rangle} ts$, iff they show the same behavior regarding predicates and updates, i.e.:
The computation $z \in \mathcal{C}^\omega$ of the HyperTSL$^-$ trace has the following relationship to the syntactically converted updates of the HyperLTL trace:

$$z(t)(\texttt{s}) = \tau_F$$

where τ_F is the unique element such that $\texttt{SynCon}([\![\texttt{s} \leftarrowtail \tau_F]\!]) \in tr(t)$. Furthermore, let $\widehat{\nu} \in (2^{\widehat{\mathcal{T}}_P})^\omega$ be the projection of tr onto $\widehat{\mathcal{T}}_P$ and let $\nu \in (2^{\mathcal{T}_P})^\omega$ such that $\widehat{\nu} = \texttt{SynCon}(\nu)$.

Additionally, we require that the input stream $\iota \in \mathcal{I}^\omega$ is such that

$$\forall t \in \mathbb{N}.\ \nu(t) = \{\tau_P \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(z, \iota, t, \tau_P)\}.$$

To determine whether a HyperTSL$^-$ trace and a HyperLTL trace are corresponding, we must check whether the input sequence and the computation that generate the HyperTSL$^-$ trace, match the encoded predicates and updates of the HyperLTL trace. This means, the input sequence and computation of the HyperTSL$^-$ trace must be such that for each time step they generate the same set of predicates that hold as on the HyperLTL trace and such that the computation assigns the same cells to function terms as the updates in the HyperLTL trace. It can be the case that one HyperLTL trace corresponds to several HyperTSL$^-$ traces since different input streams $\iota \in \mathcal{I}^\omega$ can generate the same sequence of predicate sets. A HyperTSL$^-$ trace, however, only ever has exactly one corresponding HyperLTL trace. It can also be, that for a HyperLTL trace, no corresponding HyperTSL$^-$ trace exists.

Satisfaction of a formula is checked with respect to a trace set and a trace assignment. In order to check whether a HyperTSL$^-$ formula is satisfied by the HyperTSL$^-$ trace set if and only if the formula's HyperLTL equivalent is satisfied by the HyperLTL trace set, we must check whether the trace assignments map the same trace variables to the same traces, i.e., the two trace assignments are corresponding.

Definition 5.6 (Corresponding Trace Assignments)
A HyperLTL trace assignment $\Pi_L : \mathcal{V} \to (2^{\widehat{\mathcal{T}_P}})^+ \cup (2^{\widehat{\mathcal{T}_{U/\mathtt{id}}}})$ and a HyperTSL$^-$ trace assignment $\Pi^- : \mathcal{V} \to (\mathcal{C}^\omega \times \mathcal{I}^\omega)$ are corresponding with respect to a function evaluation $\langle\cdot\rangle$, denoted by $\Pi_L \sim_{\langle\cdot\rangle} \Pi^-$, iff the same trace variables only map to traces that are corresponding, that is:

$$\forall \pi \in \mathcal{V}.\ \Pi_L(\pi) \sim_{\langle\cdot\rangle} \Pi^-(\pi)$$

A trace variable π in the Π^- trace assignment must thus map to a trace that corresponds to the trace that π maps to in the HyperLTL trace assignment Π_L.

Lemma 5.1 (Evaluation of Corresponding Traces)
Let $\tau_\pi \in \mathcal{T}_{P,\mathcal{V}} \cup \mathcal{T}_{U,\mathcal{V}}$ be a predicate term or an update and $\mathtt{SynCon}(\tau_\pi)$ the atomic proposition that results from the syntactic conversion. Let tr be a HyperLTL trace that resulted from a syntactic conversion and let ts be a HyperTSL$^-$ trace such that tr and ts are corresponding. Let T_L be a HyperLTL trace set and T^- a HyperTSL$^-$ trace set. Let Π_L and Π^- be corresponding HyperLTL and HyperTSL$^-$ trace assignments as defined by Definition 5.6. For the corresponding traces tr and ts, it holds that given a function evaluation $\langle\cdot\rangle$, they evaluate the same in their respective semantics, that is:

$$\forall t \in \mathbb{N}.\ \Pi_L[\pi \to tr], T_L, t \models \mathtt{SynCon}(\tau_\pi) \Leftrightarrow \Pi^-[\pi \to ts], T^-, t \models_{\langle\cdot\rangle} \tau_\pi$$

Proof (Evaluation of Corresponding Traces) We make a case distinction over the indexed predicate term $\tau_\pi \in \mathcal{T}_{P,\mathcal{V}} \cup \mathcal{T}_{U,\mathcal{V}}$.

- Assume $\tau_\pi \in \mathcal{T}_{P,\mathcal{V}}$ that is $\tau_\pi = \tau_{P\pi}$

$$
\begin{aligned}
& \Pi^{\text{-}}, T^{\text{-}}, t \models_{\langle\cdot\rangle} \tau_{P\pi} \\
\xLeftrightarrow{\text{Def. 3.6}} \quad & \tau_P \in \{\tau_P \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(\#1(\Pi^{\text{-}}(\pi)), \#2(\Pi^{\text{-}}(\pi)), t, \tau_P)\} \\
\xLeftrightarrow{\text{Def. 5.5}} \quad & \texttt{SynCon}(\tau_P) \in \Pi_L(\pi)\ (t) \\
\xLeftrightarrow{\text{Def. 2.5}} \quad & \Pi_L, T_L, t \models \texttt{SynCon}(\tau_{P\pi})
\end{aligned}
$$

- Assume $\tau_\pi \in \mathcal{T}_{U,\mathcal{V}}$ that is $\tau_\pi = [\![\mathtt{s} \leftarrowtail \tau_F]\!]_\pi$

$$
\begin{aligned}
& \Pi^{\text{-}}, T^{\text{-}}, t \models_{\langle\cdot\rangle} [\![\mathtt{s} \leftarrowtail \tau_F]\!]_\pi \\
\xLeftrightarrow{\text{Def. 3.4}} \quad & \#1(\Pi^{\text{-}}(\pi))\ (t)\ (\mathtt{s}) = \tau_F \\
\xLeftrightarrow{\text{Def. 5.6, 5.5}} \quad & \texttt{SynCon}([\![\mathtt{s} \leftarrowtail \tau_F]\!]) \in \Pi_L(\pi)\ (t) \\
\xLeftrightarrow{\text{Def. 2.5}} \quad & \Pi_L, T_L, t \models \texttt{SynCon}([\![\mathtt{s} \leftarrowtail \tau_F]\!]_\pi)
\end{aligned}
$$

□

We now construct a HyperTSL$^{\text{-}}$ strategy from a HyperLTL strategy as follows: An input sequence for a HyperTSL$^{\text{-}}$ strategy is denoted as ν and the corresponding input sequence for HyperLTL is denoted as $\widehat{\nu}$ where the relation $\widehat{\nu} = \texttt{SynCon}(\nu)$ holds.

Definition 5.7 (HyperTSL$^{\text{-}}$ Strategy Construction)
Let $h : (2^{\widehat{\mathcal{T}}_P})^+ \to (2^{\widehat{\mathcal{T}}_{U/\mathtt{id}}})$ be a HyperLTL strategy and $\nu(0) \dots \nu(t) \in (2^{\mathcal{T}_P})^+$ be an input sequence for a HyperTSL$^{\text{-}}$ strategy at time point t. Let $z \in \mathcal{C}^\omega$ be the computation such that

$$z(t)(\mathtt{c}) = \tau_F,$$

where $\mathtt{c} \in \mathbb{C}, t \in \mathbb{N}$, and τ_F is the unique element such that $\texttt{SynCon}([\![\mathtt{c} \leftarrowtail \tau_F]\!]) \in h(\widehat{\nu}(0) \dots \widehat{\nu}(t))$. The HyperTSL$^{\text{-}}$ strategy $f : (2^{\mathcal{T}_P})^+ \to \mathcal{C}$ is constructed from the HyperLTL strategy h as follows:

$$f(\nu(0) \dots \nu(t)) = z(t)$$

Proposition 5.1 (Corresponding Trace for a HyperTSL$^{\text{-}}$ Strategy)
Let $h : (2^{\widehat{\mathcal{T}}_P})^+ \to (2^{\widehat{\mathcal{T}}_{U/\mathtt{id}}})$ be a HyperLTL strategy and $f : (2^{\mathcal{T}_P})^+ \to \mathcal{C}$ be the HyperTSL$^{\text{-}}$ strategy that was constructed from h. It holds that for each trace in the trace set $\mathit{traces}(f)$, that is constructed as by Definition 5.1, there is a corresponding trace in the trace set $\mathit{traces}(h)$. Formally:

$$\forall\ \mathit{ts} \in \mathit{traces}(f).\ \exists\ \mathit{tr} \in \mathit{traces}(h).\ \mathit{tr} \sim_{\langle\cdot\rangle} \mathit{ts}$$

We prove that the approximation is sound and that we can indeed construct a HyperTSL$^-$ strategy from a HyperLTL strategy.

Lemma 5.2 (Soundness)
If the HyperLTL formula φ_{HLTL} that results from the syntactic conversion of a universal HyperTSL$^-$ formula φ_- is realizable, then the HyperTSL$^-$ formula φ_- is realizable as well.

Proof (Soundness) Assume φ_{HLTL} is a universal HyperLTL formula with $\varphi_{HLTL} = \texttt{SynCon}(\varphi_-)$ and φ_{HLTL} is realizable. Then there exists a winning strategy $h : (2^{\widehat{\mathcal{T}}_P})^+ \to (2^{\widehat{\mathcal{T}}_{U/\texttt{id}}})$ that realizes the HyperLTL formula. Therefore, it holds that $\Pi_\emptyset, \mathit{traces}(h) \models \varphi_{HLTL}$. We want to show that φ_- is then also realizable, meaning there exists a strategy $f : (2^{\mathcal{T}_P})^+ \to (\mathcal{C})$ such that for all function assignments $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$, the set $\mathit{traces}(f)$ satisfies the formula, that is $\Pi_\emptyset, \mathit{traces}(f) \models_{\langle\cdot\rangle} \varphi_-$. Let $\langle\cdot\rangle$ be some function assignment. We construct the strategy f from h as in Definition 5.7. In the following we denote $\mathit{traces}(f)$ by T^- and $\mathit{traces}(h)$ by T_L.

Let φ_- be of the form $\forall_1\pi_1 \dots \forall_n\pi_n.\varphi'_-$ where φ'_- is a quantifier-free formula. We construct the trace assignment Π^- from the empty assignment and based on Π_L as follows: Assume we find $\forall\pi$ at the beginning of the HyperTSL$^-$ formula prefix. We know that the HyperLTL formula has the same quantifier prefix as the HyperTSL$^-$ formula and, thus, also starts with $\forall\pi$. Let $ts \in T^-$ over $AP = \mathcal{T}_P \cup \mathcal{T}_U$ be an arbitrary HyperTSL$^-$ trace. Let $tr \in T_L$ over $AP = \widehat{\mathcal{T}}_P \cup \widehat{\mathcal{T}}_U$ be a corresponding HyperLTL trace, which exists by Proposition 5.1 , that is $tr \sim_{\langle\cdot\rangle} ts$. Since the HyperLTL strategy h is winning, π can be assigned to tr in Π_L and h still has a winning strategy for the remaining formula. We proceed with the next quantifier. This results in the trace assignments Π^- and Π_L that are corresponding as defined in Definition 5.6.

In order to prove $\Pi_\emptyset, T_L \models \varphi_{HLTL} \Rightarrow \Pi_\emptyset, T^- \models_{\langle\cdot\rangle} \varphi_-$, we prove a stronger claim:

$$\Pi^-, T^-, t \models_{\langle\cdot\rangle} \varphi_- \Leftrightarrow \Pi_L, T_L, t \models \varphi_{HLTL}$$

for all trace assignments Π^- and Π_L, where Π^- and Π_L are corresponding as by Definition 5.6.
We show by structural induction over the structure of a quantifier free formula φ_- that $\forall t \in \mathbb{N}.\ \Pi^-, T^-, t \models_{\langle\cdot\rangle} \varphi_- \Leftrightarrow \Pi_L, T_L, t \models \texttt{SynCon}(\varphi_-)$.

- Case $\varphi_- = \tau_{P\pi}$

$$\Pi^-, T^-, t \models_{\langle\cdot\rangle} \tau_{P\pi} \quad \overset{\text{Lem. 5.1}}{\Longleftrightarrow} \quad \Pi_L, T_L, t \models \texttt{SynCon}(\tau_{P\pi})$$

- Case $\varphi_- = [\![\texttt{s} \leftarrowtail \tau_F]\!]_\pi$

$$\Pi^-, T^-, t \models_{\langle\cdot\rangle} [\![\texttt{s} \leftarrowtail \tau_F]\!]_\pi \quad \overset{\text{Lem. 5.1}}{\Longleftrightarrow} \quad \Pi_L, T_L, t \models \texttt{SynCon}([\![\texttt{s} \leftarrowtail \tau_F]\!]_\pi)$$

- Case $\varphi_{\text{-}} = \neg\psi$

$$
\begin{aligned}
& & & \Pi^{\text{-}}, T^{\text{-}}, t \models_{\langle\cdot\rangle} \neg\psi \\
& \xLeftrightarrow{\text{Def. 3.4}} & & \Pi^{\text{-}}, T^{\text{-}}, t \not\models_{\langle\cdot\rangle} \psi \\
& \xLeftrightarrow{IH} & & \Pi_L, T_L, t \not\models_{\langle\cdot\rangle} \texttt{SynCon}(\psi) \\
& \xLeftrightarrow{\text{Def. 2.5, 5.2}} & & \Pi_L, T_L, t \models_{\langle\cdot\rangle} \texttt{SynCon}(\neg\psi)
\end{aligned}
$$

- Case $\varphi_{\text{-}} = \bigcirc\psi$

$$
\begin{aligned}
& & & \Pi^{\text{-}}, T^{\text{-}}, t \models_{\langle\cdot\rangle} \bigcirc\psi \\
& \xLeftrightarrow{\text{Def. 3.4}} & & \Pi^{\text{-}}, T^{\text{-}}, t+1 \models_{\langle\cdot\rangle} \psi \\
& \xLeftrightarrow{IH} & & \Pi_L, T_L, t+1 \models \texttt{SynCon}(\psi) \\
& \xLeftrightarrow{\text{Def. 2.5}} & & \Pi_L, T_L, t \models \bigcirc\texttt{SynCon}(\psi) \\
& \xLeftrightarrow{\text{Def. 5.2}} & & \Pi_L, T_L, t \models \texttt{SynCon}(\bigcirc\psi)
\end{aligned}
$$

- Case $\varphi_{\text{-}} = \psi_1 \wedge \psi_2$

$$
\begin{aligned}
& & & \Pi^{\text{-}}, T^{\text{-}}, t \models_{\langle\cdot\rangle} \psi_1 \wedge \psi_2 \\
& \xLeftrightarrow{\text{Def. 3.4}} & & \Pi^{\text{-}}, T^{\text{-}}, t \models_{\langle\cdot\rangle} \psi_1 \ \wedge \ \Pi^{\text{-}}, T^{\text{-}}, t \models_{\langle\cdot\rangle} \psi_2 \\
& \xLeftrightarrow{IH} & & \Pi_L, T_L, t \models \texttt{SynCon}(\psi_1) \ \wedge \ \Pi_L, T_L, t \models \texttt{SynCon}(\psi_2) \\
& \xLeftrightarrow{\text{Def. 2.5, 5.2}} & & \Pi_L, T_L, t \models \texttt{SynCon}(\psi_1 \wedge \psi_2)
\end{aligned}
$$

- Case $\varphi_{\text{-}} = \psi_1 \,\mathcal{U}\, \psi_2$

$$
\begin{aligned}
& & & \Pi^{\text{-}}, T^{\text{-}}, t \models_{\langle\cdot\rangle} \psi_1 \,\mathcal{U}\, \psi_2 \\
& \xLeftrightarrow{\text{Def. 3.4}} & & \exists i \geq t : \ \Pi^{\text{-}}, T^{\text{-}}, i \models_{\langle\cdot\rangle} \psi_2 \ \wedge \ \forall t \leq j < i : \ \Pi^{\text{-}}, T^{\text{-}}, j \models_{\langle\cdot\rangle} \psi_1 \\
& \xLeftrightarrow{IH} & & \exists i \geq t : \ \Pi_L, T_L, i \models \texttt{SynCon}(\psi_2) \ \wedge \\
& & & \forall t \leq j < i : \ \Pi^{\text{-}}, T^{\text{-}}, j \models \texttt{SynCon}(\psi_1) \\
& \xLeftrightarrow{\text{Def. 2.5}} & & \Pi_L, T_L, t \models \texttt{SynCon}(\psi_1) \ \mathcal{U} \ \texttt{SynCon}(\psi_2) \\
& \xLeftrightarrow{\text{Def. 5.2}} & & \Pi_L, T_L, t \models \texttt{SynCon}(\psi_1 \,\mathcal{U}\, \psi_2)
\end{aligned}
$$

□

We thus know that, if the HyperLTL approximation of the HyperTSL$^{\text{-}}$ formula is realizable, then the HyperTSL$^{\text{-}}$ formula is realizable as well.

If the HyperTSL$^-$ formula is not realizable, the tool returns a counter strategy. A counter strategy, just like a normal strategy, is an infinite tree, but instead of mapping predicate sequences to a set of computation steps, it maps a sequence of computation steps to a set of predicate terms. Formally the type of a counter strategy q is $q : \mathcal{C}^+ \to 2^{\mathcal{T}_P}$. A counter strategy for the approximation is checked for spuriousness to determine whether it is a counter strategy for the HyperTSL$^-$ formula as well. There are several scenarios in which a counter strategy for a HyperTSL$^-$ strategy can be spurious. A counter strategy for a HyperTSL$^-$ strategy can be spurious if there are two computations such that the two corresponding branches evaluate differently on equal predicate terms at two points in time. For instance: According to the strategy the predicate $\texttt{p\_x}_\pi$ holds at time point t and $\texttt{p\_x}_{\pi'}$ does not hold at time point t' even though according to the evaluation function they should evaluate the same at the respective time points. This scenario contains the case where we can have spurious behavior on only one trace, if we choose the same trace for π and π'. In addition to that, it can be that there is a branch given a computation or two branches given two computations such that two predicate terms that are constructed using the same predicate but that take different arguments, evaluate differently according to the strategy, even though the arguments were the same when evaluated at two points in time. For instance: According to the strategy $\texttt{p\_x}_\pi$ holds at time point t on trace π and $\texttt{p\_y}_\pi$ does not hold at time point t even though `x` and `y` have the same value and predicates should always evaluate the same under equal values. Combining the above, a counter strategy q is *spurious* if there are two branches such that two equal predicate terms evaluate differently according to the strategy at two points in time.

Example 5.4 (Spurious Counter Strategy)
A spurious counter strategy for our example formula as stated in Example 5.1 would be to always have `p_y` hold on a trace, while on the same trace `p_x` never holds. This makes the HyperLTL formula, as stated in Example 5.3 evaluate to false, and thus causes an unrealizability result. But there is a computation where we can update `x` with `y` as soon as `p y` holds. We see that the counter strategy is spurious in this scenario, as `x` and `y` are the same now and, thus, the predicate should evaluate the same on equal values.

The set of predicates that holds at a certain time point on a branch in a strategy tree, depends on the input stream and computation that belong to this branch. We define when an input stream matches a computation and a strategy as follows:

Definition 5.8 (Matching Input Streams)
An input stream $\iota \in \mathcal{I}^\omega$ *matches* the branch created by a strategy $q : \mathcal{C}^+ \to 2^{\mathcal{T}_P}$ under a computation $\sigma \in \mathcal{C}^\omega$, denoted by $q \wr\wr (\sigma, \iota)$, iff

$$\forall t \in \mathbb{N}.\ q(\sigma(0) \dots \sigma(t)) = \{\tau_P \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(\sigma, \iota, t, \tau_P)\}$$

We express spuriousness of a HyperTSL$^-$ counter strategy as follows:

Definition 5.9 (Hyper-Spuriousness)
We define a property $\texttt{pred} : \mathcal{T}_{P,\mathcal{V}} \to \mathcal{T}_{P,\mathcal{V}} \to \mathbb{B}$ that checks whether two predicate terms are constructed from the same predicate under potentially different arguments and on potentially different traces.

$$\begin{aligned}
&\texttt{pred}\ ([\texttt{p}\ \tau_{F_1}\ \ldots \tau_{F_n}]_\pi)\ ([\texttt{p}\ \tau'_{F_1}\ \ldots \tau'_{F_n}]_{\pi'}) && = \top \\
&\texttt{pred}\ _____ && = \bot
\end{aligned}$$

In the following, we use τ_π to denote a predicate term of the form $[\texttt{p}\ \tau_{F_1}\ \ldots \tau_{F_n}]_\pi$ and τ without index if we mean the respective term but without index, that is $\texttt{p}\ \tau_{F_1}\ \ldots \tau_{F_n}$. A counter strategy $q : \mathcal{C}^+ \to 2^{\mathcal{T}_P}$ for HyperTSL$^-$ is spurious if the following holds:

$$\begin{aligned}
&\exists\ \sigma_1, \sigma_2 \in \mathcal{C}^\omega.\ \exists\ \iota_1, \iota_2 \in \mathcal{I}^\omega.\ \exists\ t, t' \in \mathbb{N}.\ \exists \tau_\pi, \tau'_{\pi'} \in \mathcal{T}_{P,\mathcal{V}}. \\
&\quad \texttt{pred}\ \tau_\pi\ \tau'_{\pi'} \\
&\wedge\ \tau \in q(\sigma_1(0)\ \ldots\ \sigma_1(t-1)) \wedge \tau' \notin q(\sigma_2(0)\ \ldots\ \sigma_2(t'-1)) \\
&\wedge\ \forall \langle\cdot\rangle : \mathbb{F} \to \mathcal{F}.\ q \wr\wr\ (\sigma_1, \iota_1)\ \wedge\ q \wr\wr\ (\sigma_2, \iota_2)\ \wedge\ \eta_{\langle\cdot\rangle}(\sigma_1, \iota_1, t, \tau) = \eta_{\langle\cdot\rangle}(\sigma_2, \iota_2, t', \tau')
\end{aligned}$$

The first conjunct checks that the two predicate terms we consider are constructed from the same predicate. The second conjunct checks whether the two predicate terms evaluate differently according to the strategy at any two time points, i.e., one predicate holds and the other one does not. The first two conjuncts $q \wr\wr (\sigma_1, \iota_1)$ and $q \wr\wr (\sigma_2, \iota_2)$ after the introduction of the evaluation function by a universal quantifier in the last line, ensure that the input streams we consider match the computations and in consequence cause the same sets of predicates to hold as specified by the counter strategy. This check is necessary to provide the evaluation function $\eta_{\langle\cdot\rangle}$ with the input streams that belong to the computation, and not just any inputs streams, as the matching ones are needed to evaluate the two predicate terms. Finally, the last conjunct checks whether the evaluation function would actually evaluate the predicates equally. Note that this definition covers all previously mentioned cases, as the predicate `pred` covers the case where the predicate terms are exactly the same, but also where they take different arguments as well as the fact that they can stem from different traces. Since the definition introduces the computations σ_1 and σ_2 using an existential quantifier, for the case where we only check spuriousness of a single branch, we can just choose the same computation both times.

If the check evaluates to true we know that the counter strategy is spurious. This is because when using HyperLTL as approximation language, the semantics of predicates get lost. Especially the fact that a predicate should always evaluate to the same value when given the same argument gets lost and is, as we have seen, the reason for spuriousness. This result indicates that there is still a chance for the formula to be realizable. We, therefore, need to add constraints to the approximation of the formula

in order to restore the desired behavior of predicates, which is to always evaluate the same on equal arguments. These additional constraints restrict the atomic propositions such that they present the predicates more accurately and eliminate the previously found spurious behavior. When trying to eliminate the spurious behavior caused by the counter strategy we described in Example 5.4, intuitively this specific spurious counter strategy can be avoided by adding the formula $\forall\pi.\ \texttt{p\_y}_\pi \wedge \texttt{x\_up\_y}_\pi \rightarrow \texttt{p\_x}_\pi$ to the approximation as stated in Example 5.3. This way, it is ensured that if $\texttt{p\_y}_\pi$ holds and the cell `x` was updated with `y` on trace π, then $\texttt{p\_x}_\pi$ also holds. This refinement is not constructed automatically and there might be further spurious counter strategies even after adding this refinement. After the refinement, we again try to construct a realizing strategy for the formula again. This procedure is repeated until we either get a realizability result, or until we find a non-spurious counter strategy, or until the predefined bound is reached. If a non-spurious counter strategy is found, this means that the formula is indeed unrealizable and the synthesis process terminates.

Chapter 6

Conclusion

In this thesis, we constructed a hyperlogic for temporal stream logic, HyperTSL, for expressing hyperproperties about infinite data and identified a fragment of it, called HyperTSL$^-$. Both HyperTSL and its fragment HyperTSL$^-$ extend TSL with explicit quantification over traces and allow expressing hyperproperties on systems described in TSL. We defined syntax and semantics and an indexed term notation for HyperTSL and HyperTSL$^-$. HyperTSL$^-$ is a syntactic fragment of HyperTSL, which differs in how predicate terms are indexed. For HyperTSL, the arguments of predicate terms can be seen as individual components and are indexed individually, and for HyperTSL$^-$, the predicate and its arguments are seen as one component. The arguments of a predicate in a HyperTSL formula can thus be from different traces, while in a HyperTSL$^-$ formula the arguments of a predicate must all come from the same trace. Furthermore, we used the hyperlogics to express hyperproperties on two systems described in TSL. While the first system was rather small to illustrate the use of the hyperlogic, we also presented a larger use case in Chapter 4. There, we specified hyperproperties on an exhaust emission control system in order to spot doped software. The use case was inspired by the Diesel scandal in 2015. In Chapter 5, we presented a bounded synthesis approach for the universal fragment of HyperTSL$^-$. The approach we presented is sound, which we showed in Proof 5.2.

HyperTSL is more expressive than HyperLTL. HyperTSL allows expressing hyperproperties involving infinite data, as it abstracts from concrete data using predicates and functions. Expressing these hyperproperties is not possible using HyperLTL. Furthermore, as HyperTSL uses predicates, functions, and updates, it allows an intuitive description of the control flow in a system. As we have seen, it can express many useful hyperproperties in the field of software doping, e.g., robust cleanness, and there are more examples of software doping where it can be used. While synthesis for HyperTSL and HyperTSL$^-$ is undecidable, we could still provide a sound bounded synthesis approach for universal HyperTSL$^-$.

6.1 Future Work

For future work, the first step would be to extend the theoretical bounded synthesis approach we presented to also work for existential HyperTSL$^-$ formulas. Afterward, the approach could be implemented and used to synthesize formulas expressed in HyperTSL$^-$. Furthermore, it would be interesting to explore HyperTSL synthesis and if possible provide an approach as well. A first challenge for HyperTSL synthesis lies in characterizing realizability for HyperTSL. If existing HyperLTL tools should be exploited like for HyperTSL$^-$ synthesis, another challenge in specifying a synthesis approach for HyperTSL lies in finding a suitable HyperLTL encoding for predicate terms that take arguments from different traces. Predicate terms that take arguments from different traces would need to be encoded as atomic propositions in a way that allows re-inferring from what trace the individual arguments came. Moreover, the encoding should be sound. Perhaps an approximation language other than HyperLTL could be found.

Another interesting topic which we did not cover in this thesis is the model checking problem for both HyperTSL and HyperTSL$^-$. In model checking, an already existing system is checked against a specification. Model checking is especially interesting for HyperTSL specifications that cannot be synthesized using the bounded synthesis approach. As the synthesis approach is not complete, there might still exist a system that realizes the formula. If such a system is constructed using a different approach, model checking can be used to determine whether it really fulfills the specification given in HyperTSL. The first question that comes up is how to treat functions and predicates in HyperTSL model checking because in TSL and HyperTSL, functions and predicates are uninterpreted. If for HyperTSL model checking functions and predicates should already have an interpretation, we need to include theories in the logic. TSL modulo theories was explored by Heim in [13] and can function as a starting point for HyperTSL modulo theories. The decidability results that he established for TSL modulo theories present a lower bound for the decidability of HyperTSL modulo theories. The next challenge for HyperTSL model checking would be to define automata that can represent the HyperTSL formula.

Bibliography

[1] Célia Alves, Ana Calvo, Diogo Lopes, Teresa Nunes, Aurélie Charron, M. Goriaux, P. Tassel, and Pascal Perret. Emissions of Euro 3-5 Passenger Cars Measured Over Different Driving Cycles. 2013. doi:10.13140/2.1.1995.1048.

[2] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal Logics for Hyperproperties. *CoRR*, abs/1401.4492, 2014. URL `http://arxiv.org/abs/1401.4492`.

[3] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying Hyperliveness. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 121–139. Springer, 2019. doi:10.1007/978-3-030-25540-4_7.

[4] Pedro R. D'Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. Is your Software on Dope? Formal Analysis of surreptitiously "enhanced" Programs. *CoRR*, abs/1702.04693, 2017. URL `http://arxiv.org/abs/1702.04693`.

[5] Dieselnet. ECE 15 + EUDC / NEDC, 2013. URL `https://dieselnet.com/standards/cycles/ece_eudc.php`. [Online; accessed 22-October-2020].

[6] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986. doi:10.1145/4904.4999.

[7] Bernd Finkbeiner. Synthesis of Reactive Systems. In *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 72–98. IOS Press, 2016. doi:10.3233/978-1-61499-627-9-72.

[8] Bernd Finkbeiner. Temporal Hyperproperties. *Bull. EATCS*, 123, 2017. URL `http://eatcs.org/beatcs/index.php/beatcs/article/view/514`.

[9] 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, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015. doi:10.1007/978-3-319-21690-4_3.

[10] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal Stream Logic: Synthesis beyond the Bools. *CoRR*, abs/1712.00246, 2017. URL `http://arxiv.org/abs/1712.00246`.

[11] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. Synthesis from hyperproperties. *Acta Informatica*, 57(1-2):137–163, 2020. doi:10.1007/s00236-019-00358-2.

[12] Georgios Fontaras, Vicente Franco, Panagiota Dilara, Giorgio Martini, and Urbano Manfredi. Development and review of euro 5 passenger car emission factors based on experimental results over various driving cycles. *Science of The Total Environment*, 468-469:1034 – 1042, 2014. ISSN 0048-9697. doi:https://doi.org/10.1016/j.scitotenv.2013.09.043.

[13] Philippe Heim. Satisfiability of Temporal Stream Logic modulo Theories. Bachelor Thesis at Saarland University, 2020.

[14] J. E. Jonson, J. Borken-Kleefeld, D. Simpson, A. Nyíri, M. Posch, and C. Heyes. Impact of excess NO x emissions from diesel cars on air quality, public health and eutrophication in Europe. *Environmental Research Letters*, 12(9):094017, 2017. doi:10.1088/1748-9326/aa8850.

[15] Rik Oldenkamp, Rosalie van Zelm, and Mark A.J. Huijbregts. Valuing the human health damage caused by the fraud of Volkswagen. *Environmental Pollution*, 212:121 – 127, 2016. ISSN 0269-7491. doi:https://doi.org/10.1016/j.envpol.2016.01.053.

[16] Per Kågeson. Cycle-Beating and the EU TestCycle for Cars. Technical report, European Federation for Transport an Environment, 1998. URL `https://www.transportenvironment.org/sites/te/files/media/T&E%2098-3_0.pdf`.

[17] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.32.

[18] Tagesschau. Konzern gibt Zahlen bekannt - Fünf Millionen Autos der Marke VW betroffen, 2015. URL `https://www.tagesschau.de/wirtschaft/vw-nutzfahrzeuge-107.html`. [Online; accessed 23-September-2020].

[19] Tagesschau. Illegale Abschalteinrichtung beim Golf 7?, 2020. URL `https://www.tagesschau.de/investigativ/swr/vw-abgasskandal-161.html`. [Online; accessed 25-October-2020].