Logical and Deep Learning Methods for Temporal Reasoning

A dissertation submitted towards the degree Doctor of Natural Sciences (Dr. rer. nat.) of the Faculty of Mathematics and Computer Science of Saarland University

Christopher Hahn

Saarbrücken
2021

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor Prof. Bernd Finkbeiner, Ph.D.: Dear Bernd, I thank you for countless productive scientific discussions and our many teaching endeavors. Especially, I appreciate your trust in me when I felt the need to explore new paths. I sincerely thank you for everything you've done for me.

I am fortunate that Prof. Dr. Andreas Podelski and Dr. Christian Szegedy kindly agreed to review this thesis. I would like to thank you both for your outstanding contributions in your respective fields, for your helpful comments, and for taking the time to review this thesis.

Special thanks go to my co-authors of the publications whose content made it into this thesis: Norine Coenen, Bernd Finkbeiner, Jens U. Kreber, Yannick Schillo, Markus N. Rabe, Leander Tentrup, and, especially, Jana Hofmann and Frederik Schmitt.

I would also like to thank the following people: my office mates Jana Hofmann and Hazem Torfah, with whom I shared plenty of awesome memories and numerous publications; my co-authors on other projects Tamara Flemisch, Tobias Hans, Tom Horak, Philip Lukert, Niklas Metzger, and Marvin Stenger; my colleagues Alexander, Christa, Felix, Florian, Hadar, Jan, Jesko, Julian, Malte, Martin, Maximilian, Michael, Mouhammad, Noemi, Peter, Sabine and Swen; the students and all the student TAs I worked with, especially to everyone involved in Prog1'17; and, last but not least, my fellow students and friends Clara, Joris, Sebastian, and Yannick.

To my parents, my siblings, the rest of my family, and to everyone that has supported me throughout my life: Thank you. To André: I am truly grateful for your support over all these years. You are a great friend, thank you. To Sophie and Charlotte: You cannot imagine how happy you make me day by day. I am incredibly grateful seeing you grow up and explore the world.

Last but not least. Dear Nadine, I would like to thank you from the bottom of my heart for your unconditional support over the course of writing this thesis. I thank you for your love, your patience, your encouragement, and for "Dein Hurra in mein Gesicht und alle Zweifel, alle Faxen werden lächerlich" *(Bosse – Dein Hurra)*.

Abstract

In this thesis, we study logical and deep learning methods for the temporal reasoning of reactive systems.

In Part I, we determine decidability borders for the satisfiability and realizability problem of temporal hyperproperties. Temporal hyperproperties relate multiple computation traces to each other and are expressed in a temporal hyperlogic. In particular, we identify decidable fragments of the highly expressive hyperlogics HyperQPTL and HyperCTL*. As an application, we elaborate on an enforcement mechanism for temporal hyperproperties. We study explicit enforcement algorithms for specifications given as formulas in universally quantified HyperLTL.

In Part II, we train a (deep) neural network on the trace generation and realizability problem of linear-time temporal logic (LTL). We consider a method to generate large amounts of additional training data from practical specification patterns. The training data is generated with classical solvers, which provide one of many possible solutions to each formula. We demonstrate that it is sufficient to train on those particular solutions such that the neural network generalizes to the semantics of the logic. The neural network can predict solutions even for formulas from benchmarks from the literature on which the classical solver timed out. Additionally, we show that it solves a significant portion of problems from the annual synthesis competition (SYNTCOMP) and even out-of-distribution examples from a recent case study.

Zusammenfassung

Diese Arbeit befasst sich mit logischen Methoden und mehrschichtigen Lernmethoden für das zeitabhängige Argumentieren über reaktive Systeme.

In Teil I werden die Grenzen der Entscheidbarkeit des Erfüllbarkeits- und des Realisierbarkeitsproblem von temporalen Hypereigenschaften bestimmt. Temporale Hypereigenschaften setzen mehrere Berechnungsspuren zueinander in Beziehung und werden in einer temporalen Hyperlogik ausgedrückt. Insbesondere werden entscheidbare Fragmente der hochexpressiven Hyperlogiken HyperQPTL und HyperCTL* identifiziert. Als Anwendung wird ein Enforcement-Mechanismus für temporale Hypereigenschaften erarbeitet. Explizite Enforcement-Algorithmen für Spezifikationen, die als Formeln in universell quantifiziertem HyperLTL angegeben werden, werden untersucht.

In Teil II wird ein (mehrschichtiges) neuronales Netz auf den Problemen der Spurgenerierung und Realisierbarkeit von Linear-zeit Temporallogik (LTL) trainiert. Es wird eine Methode betrachtet, um aus praktischen Spezifikationsmustern große Mengen zusätzlicher Trainingsdaten zu generieren. Die Trainingsdaten werden mit klassischen Solvern generiert, die zu jeder Formel nur eine von vielen möglichen Lösungen liefern. Es wird gezeigt, dass es ausreichend ist, an diesen speziellen Lösungen zu trainieren, sodass das neuronale Netz zur Semantik der Logik generalisiert. Das neuronale Netz kann Lösungen sogar für Formeln aus Benchmarks aus der Literatur vorhersagen, bei denen der klassische Solver eine Zeitüberschreitung hatte. Zusätzlich wird gezeigt, dass das neuronale Netz einen erheblichen Teil der Probleme aus dem jährlichen Synthesewettbewerb (SYNTCOMP) und sogar Beispiele außerhalb der Distribution aus einer aktuellen Fallstudie lösen kann.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In computer science, a (computer) system that continuously interacts with its environment is called a *reactive system*. Examples are hardware circuits, communication protocols, or embedded controllers [106]. They are indispensable in today's industry and are deployed in many safety and security-critical systems, such as cars, aircraft, or the Internet of Things (IoT). The failure of a safety-critical system could result in fatalities, and daily security breaches of security-critical systems lead to considerable economic damage. As a result, global spending on cybersecurity is expected to exceed the 1 trillion mark by 2035 [90]. The quality standard set for a reactive system deployed in such critical environments is exceptionally high. Companies like Airbus, Intel, or Amazon, thus increasingly rely on quality assurance measures that mathematically prove that a reactive system behaves as expected. Airbus uses Absint's [3] abstract interpretation[1] tools as part of its certification process. Intel replaced testing with verification for the core execution cluster in their design of the Intel Core i7 processor [120]. Recently, the initial boot code in data centers at Amazon Web Services (AWS) has been checked to be memory safe [45]. All these approaches have in common that the expected behavior of the system must be rigorously specified.

A *(temporal) specification* of a reactive system precisely describes, given a particular input, what the output at a given point in time should be. When being interested in *reasoning* on the *temporal* behavior of a reactive system, we must provide specifications that are even stricter than just being precise: a *formal* specification follows a syntax and, especially, a formal semantics. Therefore, it cannot be based on natural language, which is inherently unclear and ambiguous, but must be based on rigorous mathematics. *Temporal logics* are a widespread family of formal specification

[1]Abstract interpretation [46] is a method to approximate the semantics of computer systems soundly.

languages for temporal reasoning. Their foundation dates back to Tense Logic [170] introduced by Arthur Prior in his 1955/1956 John Locke lecture [6] and extended by Hans Kamp in 1968 in his Ph.D. thesis [123]. In 1977, Amir Pnueli first proposed this specification paradigm for computer science in "The Temporal Logic of Programs" [165].[2] *Linear-time Temporal Logic* (LTL) uses the temporal modalities of Tense Logic, i.e., U (until), $\square$ (globally), $\diamondsuit$ (eventually), and $\bigcirc$ (next) to specify the behavior of a reactive system. For example, one can express that “at no point in time access to a shared resource should be given until the resource is free” as follows: $\square((\neg \mathit{access})\, \mathcal{U} \mathit{free})$. This paradigm provides an approach to express intuitive yet rigorously defined specifications of reactive systems (see Section 1.2 and Chapter 2 for detailed examples). The development of algorithms for temporal specifications was enabled by identifying the relation between temporal logics and finite automata. This connection has been examined by Moshe Vardi and Pierre Wolper in “An Automata-theoretic Approach to Automatic Program Verification” [211, 212, 218].[3] A temporal logical formula can be translated into a finite automaton with a Büchi acceptance condition (named after Julius Richard Büchi) [30]. The Büchi automaton accepts an infinite word if at least one accepting state is visited infinitely often. Such finite automata over infinite words are one of the cornerstones in industry-strength verification tools used by, e.g., the companies mentioned above.

Modern circumstances, however, pose new challenges for the reasoning over reactive systems with temporal logics.

The increased linkage of computer systems through the Internet and the growth of the IoT in general results in numerous security vulnerabilities, such as the infamous Meltdown [145] and Spectre [126] attacks. Classical temporal logics, such as abovementioned LTL, cannot express *information-flow (security) policies*. Information-flow policies restrict the flow of confidential data through a reactive system, such that attackers gain no information by observing differences in multiple runs of a system. The general class of properties that relate multiple runs of a reactive system are called *temporal hyperproperties* [69, 41]. A variety of temporal hyperlogics were introduced to express such temporal hyperproperties (see [42] for an overview). The first challenge considered in this thesis, is to find a sweet spot between expressive power and feasibility. It is necessary to determine how much expressive power can be added to a temporal hyperlogic before running into undecidability. In Part I of this thesis, we, thus, determine the decidability boundaries of the satisfiability and synthesis problem of temporal hyperproperties. As an application of these problems, we consider the runtime enforcement problem of temporal hyperproperties.

Another challenge under modern circumstances is that reactive systems con-

[2]Amir Pnueli received the Turing Award in 1996 “For seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification.” [1].

[3]The authors received the Gödel Prize in 2000 for their contributions to temporal logics and finite automata [4].

stantly increase in size, such that classical techniques must be equipped with fast and reliable heuristics to become feasible. A current technique that yields heuristics and even state-of-the-art performance in many fields (e.g., [107, 201, 222, 155, 193]) is the training of a *(deep) neural network* [98]. So far, there was the perception that deep neural networks cannot reliably solve complex logical tasks end-to-end. With architectural advances, especially the Transformer architecture [214], however, it was demonstrated that neural networks perform surprisingly well on symbolic integration [134], and that self-supervised training leads to mathematical reasoning abilities [172]. The second challenge considered in this thesis, is to train a deep neural network on temporal logical reasoning tasks. With this approach, the temporal logic serves as a link between natural language and the benefit of verifiable predictions. Because a temporal logic has a formally defined semantics, predictions of the neural network can be automatically verified. The verification of the predictions takes a fraction of the time that would be needed to compute them classically (see Section 1.4 for more details). Training a deep neural network on a temporal logical reasoning task, thus, automatically yields a reliable heuristic for safety-critical and privacy-critical reactive systems. In Part II of this thesis, we successfully train a deep neural network on the task of generating solutions to temporal logical formulas. We provide a data generation method that even enables the successful training of a neural network to solve the LTL synthesis task, i.e., the challenging task of constructing a satisfying circuit directly out of a temporal specification.

The remainder of this introduction is structured as follows. We give an overview of the traditional problems in the field of formal methods in Section 1.2. Furthermore, we provide an example of a reactive system model and a temporal logical specification. In Section 1.3, we provide an introduction on temporal hyperproperties. We give an overview on Part I of this thesis and introduce the hyperlogics HyperQPTL, and HyperCTL*. In Section 1.4, we provide an introduction on deep learning for temporal logics. We give an overview on Part II of this thesis and introduce our training method. Section 1.5 lists the publications which this thesis is based on. In Section 1.6, we list the contributions of this thesis and, lastly, Section 1.7 provides an overview over related work.

1.2 Formal Methods

The branch of computer science that is, amongst other things[4], concerned with techniques for the formal specification and the development of reactive systems provably complying with such formal specifications is called *formal methods*. One of the

[4]The range of system models considered in the area of formal methods is broad: e.g., software models, probabilistic models, or machine learning models. This thesis focuses on the classical reactive systems (see Chapter 2 for a formal definition).

$\Box\neg(g_0 \wedge g_1)$

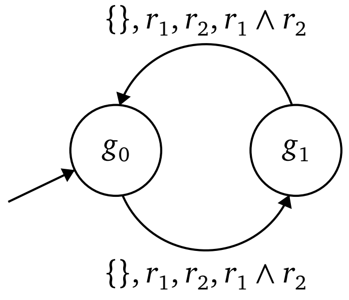

$\{\}, r_1, r_2, r_1 \wedge r_2$
g_0
g_1
$\{\}, r_1, r_2, r_1 \wedge r_2$

Figure 1.1: A mutual exclusion specification in LTL (left) and a finite state machine model of a controller (right), where request and grant is abbreviated by r and g respectively. Multiple labels on the edges denote that all of them are accepted.

most desirable objectives in the development of algorithms for formal methods is *automation* to support developers. *Verification* algorithms, for example, so-called model checkers [39], prove that an implementation adheres to a particular specification. *Synthesis* algorithms and *satisfiability* solvers are purely logical methods that do not receive a system model as input. *Synthesis* algorithms construct an implementation directly out of the specification. *Satisfiability* solvers analyze the specification at hand. They determine if it is satisfiable, and they are also used to analyze implications between specifications. *Dynamic verification* techniques can treat the reactive system as a black box and analyze its behavior during runtime. A monitor alerts the user when a run of the reactive system does not comply with the specification, whereas an enforcement algorithm corrects the system's behavior automatically. This thesis focuses on logical problems where no implementation of the reactive system is passed as input, namely satisfiability, synthesis and enforcement.

Consider, for example, a controller that manages a shared resource. It receives requests and provides grants for the resource to different processes. The LTL formula in Figure 1.1 (left) defines a mutual exclusion specification, i.e., that two processes should not have access to the shared resource simultaneously. The formula reads as "at every point in time ($\Box$) the controller should not give grants to process zero and process one simultaneously".

To apply formal methods to a reactive system, it must be modeled mathematically as well, which is typically done as some finite state machine. Figure 1.1 (right) depicts a model of a controller satisfying the sample specification. The states (circles) are labeled with the system's output, and the edges (arrows) are labeled with possible inputs from the environment. The controller provides a grant to process zero and process one alternatingly *regardless* of the input. An LTL formula reasons over (execution) traces of these state machines in discrete time, which are, as an abstraction, typically assumed to be infinite sequences of valuations of system variables. A trace thus represents the input-output behavior of a single run through a reactive system. Depending on the input from the environment, the controller in Figure 1.1 has infinitely many traces. An example trace could look as follows:

$$\{g_0\} \cdot \{r_1, g_1\} \cdot \{g_0\} \cdot \{g_1\} \cdot \{g_0\} \cdot \{g_1\} \cdots$$

In the first position of the trace (corresponding to the starting state of the state machine), the system outputs g_0, and no input is coming from the environment, i.e., there is no request to the shared resource. In the second position, the system switches states by taking the $\{\}$-transition. The system outputs g_1 and receives a request from process 1 (r_1). Finally, the output alternates between g_1 and g_0 regardless of the absence of requests after the second position. We can now verify that the controller satisfies the mutual exclusion property, by verifying that every trace of the controller satisfies the specification. While this controller satisfies the mutual exclusion specification, the controller provides many spurious grants. In Chapter 2, we define the necessary preliminaries on reactive systems and temporal logics required for this thesis in more detail.

1.3 Temporal Hyperproperties

Recently, many security breaches, most famously Meltdown [145] and Spectre [126], occurred. Meltdown “breaks the most fundamental isolation between user applications and the operating system. This attack allows a program to access the memory, and thus also the secrets, of other programs and the operating system.” [2]. This attack exploits an *information leak*. In the following explanation, we assume that there are two types of inputs and outputs: “high(-security)” inputs and outputs are only visible by an admin, and “low(-security)” inputs and outputs are observable by everyone and, hence, also by an attacker. Information leaks are hard to detect. If specific data must be kept secret, then *two* executions that result from different values of the high-security inputs (but agree on the low-security inputs) should not reveal any difference in the low-security outputs. Figure 1.2 sketches a black box reactive system with an information leak. If such a leak happens, an attacker might gain knowledge about confidential data, such as credit card information.

LTL[5] can express no information-flow policies and, in general, no properties that reason over *multiple* execution traces. Properties that relate multiple traces are called *hyperproperties* [41]. Hyperproperties generalize trace properties, which are sets of traces, to *sets of sets of traces*. While the primary motivation to study hyperproperties stems from secure information-flow control, there are many more interesting hyperproperties.

[5]As well as CTL/CTL* and all temporal logics that only reason about a trace in isolation.

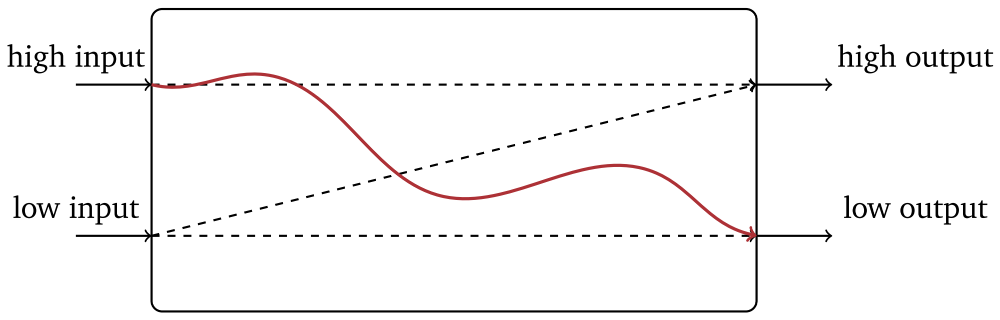


Figure 1.2: A sketch of an information leak in a black box reactive system. Dashed arrows visualize permissible information flow. The red arrow from high security input to low security output visualizes an *unwanted* flow of information.

For example:

- robustness [200] in machine learning, i.e., two inputs that are sufficiently “close” to each other should produce the same predictions,
- distributivity [167], i.e., some outputs should be independent of specific processes,
- promptness [131], i.e., there should exist a common bound across all traces for an event to occur,
- or symmetry [82], i.e., the reactive system should behave symmetrically when swapping inputs on two traces.

In the following, we provide an overview over Part I of this thesis. We describe the hyperlogics considered in this thesis and describe the enforcement problem of temporal hyperproperties.

Recently, a variety of temporal logics for hyperproperties, starting in 2014 with HyperLTL and HyperCTL* [40], have been proposed (e.g., [157, 49, 171]). HyperLTL extends LTL with a mechanism to quantify over trace variables and label propositions with these trace variables. A classical information flow policy that forbids unwanted information flow from high-security variables to low-security variables is *noninterference* [97], which can be expressed in HyperLTL as follows:

$$\forall \pi. \forall \pi'. \square(\mathit{lowinput}_\pi \leftrightarrow \mathit{lowinput}_{\pi'}) \rightarrow \square(\mathit{lowoutput}_\pi \leftrightarrow \mathit{lowoutput}_{\pi'}) \ .$$

The formula states that for all traces π and π' that agree on all points in time on the low-security input ($\square(\mathit{lowinput}_\pi \leftrightarrow \mathit{lowinput}_{\pi'})$), the traces must agree on the low-security output as well ($\square(\mathit{lowoutput}_\pi \leftrightarrow \mathit{lowoutput}_{\pi'})$). For example, an attacker could start any two runs of the reactive system with the same low-security input but

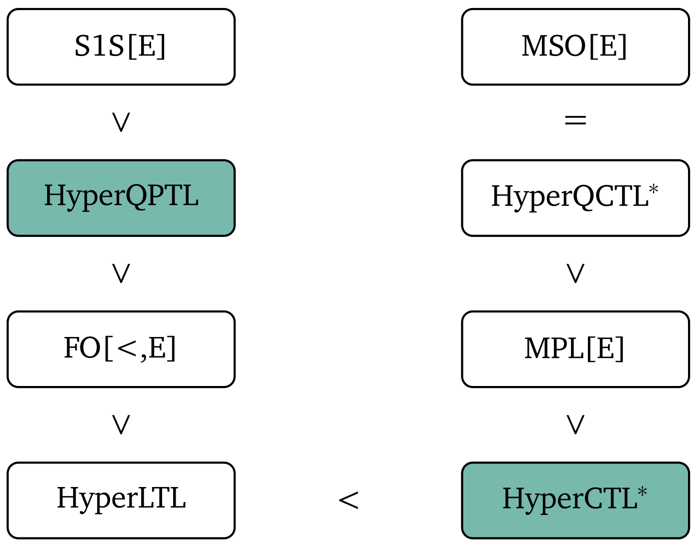


Figure 1.3: The expressiveness hierarchy of hyperlogics [42] for linear-time logics (left) and branching-time logics (right). Logics studied in the first part of this thesis are highlighted in green.

cannot distinguish the traces by its observational behavior. The primary method for verifying reactive systems against hyperproperties is by verifying the *self-composition* of copies of the system [14] (see Chapter 2 for more details). With this technique, the verification of hyperproperties reduces to the verification of trace properties over a tuple of traces.

When considering purely logical methods (e.g., satisfiability or realizability), however, there exists no implementation to self-compose. There is no clear solution on how to handle logical methods for hyperproperties. They suffer from higher computational costs and quickly become undecidable when the temporal logic is equipped with too much expressive power. Studying decidability boundaries is, thus, essential.

Fragments of HyperLTL have been identified for which logical methods remain decidable. HyperLTL formulas in the $\exists^*\forall^*$ fragment have a decidable satisfiability problem [70], while the complete logic is undecidable [70, 89]. HyperLTL synthesis can be decided for the $\exists^*\forall^1$ and the so-called *linear* $\forall^*$ fragment. The linear fragment allows only linear distributed architectures, where no information forks can be present [74]. We will discuss information forks in more detail in Chapter 3 when considering logical methods for HyperQPTL. We distinguish between linear-time hyperlogics, which are interpreted over execution traces, and branching-time hyperlogics interpreted over trees [209]. Figure 1.3 depicts the known results from a first expressiveness study of hyperlogics [42]. So far, hyperlogics were either derived from temporal logics by adding trace quantification or from first-order or second-order logics by adding an equal-level predicate, which relates the same point in time on different traces. This thesis focuses on hyperlogics derived by temporal logics. We climb up in the hierarchy of hyperlogics to determine the decidability border.

In Chapter 3, we study logical methods for HyperQPTL. HyperQPTL extends HyperLTL with quantification over sequences of new propositions. What makes the logic particularly expressive is that the trace quantifiers and propositional quantifiers can be freely interleaved. With this mechanism, HyperQPTL can not only express all ω-regular properties over a sequence of n-tuples; it truly interweaves trace quantification and ω-regularity. For example, promptness can be stated as the following HyperQPTL formula:

$$\exists b.\forall \pi.\ \Diamond b \land (\neg b \ \mathcal{U} e_\pi)\ .$$

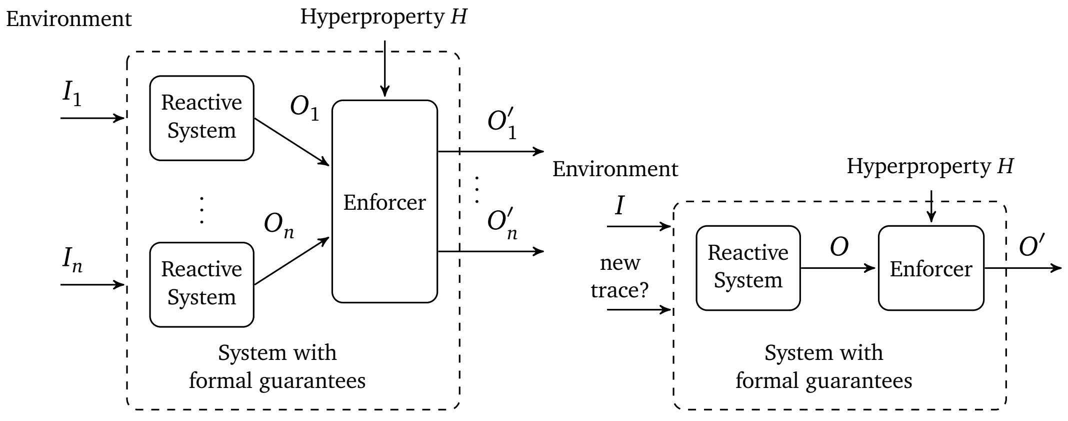
The formula states that there exists a sequence $s \in (2^{\{q\}})^\omega$, such that event e holds on all traces before the first occurrence of b in s.

In Chapter 4, we study logical methods for HyperCTL*. While linear-time temporal logics like LTL describe properties of individual traces, branching-time temporal logics like CTL and CTL* describe properties of computation trees, where the branches can be inspected by quantifying existentially or universally over paths. In contrast to HyperLTL, where quantifiers are only allowed in a *prefix*, HyperCTL* allows for arbitrary quantifiers in a formula. This expressive power is needed to state that a system can generate secret information [81]. Generating secret information means that there is, at some point, a branching into observably equivalent paths that differ in the values of a secret. For example, this property can be stated as the following HyperCTL* formula:

$$\exists \pi. \Diamond \exists \pi'. (\Box \bigwedge_{a \in P} a_\pi \leftrightarrow a_{\pi'}) \land (\bigcirc \bigvee_{a \in S} a_\pi \not\leftrightarrow a_{\pi'})\ ,$$

where the set of atomic propositions divides into the two disjoint sets of publicly observable propositions P and secret propositions S.

Logical methods for hyperproperties are essential when there is no access to the implementation to apply verification methods. In Chapter 5, we consider a practical problem to apply logical methods for hyperproperties to: the *enforcement* of hyperproperties. Enforcement mechanisms treat the system under consideration as a black-box and correct the behavior during runtime such that it complies with a given hyperproperty. It is an extension of the monitoring problem, where a monitor only reports whether an error is detected. When considering hyperproperties, especially information-flow policies, a reactive system cannot simply be stopped when an error occurs since this could lead to the attacker gaining information about a secret. It is thus vital to keep the system alive by correcting its behavior. Dynamic verification methods are especially challenging when considering hyperproperties since we have to relate *multiple* traces to each other. We study the enforcement problem



(a) Parallel model. (b) Sequential model.

Figure 1.4: Runtime enforcement for a reactive system. In case the input-output-relation would violate the hyperproperty H, the enforcer corrects the output.

under two different trace input models: The *parallel input model*, where the enforcer observes traces in parallel (for example, by applying secure multi execution [48]); moreover, in the *sequential input model*, where the enforcer observes traces one after another (for example, by observing multiple sessions). Figure 1.4 shows an overview of enforcement mechanisms for hyperproperties in both input models. We develop concrete algorithms and provide experiments for specifications given as universally quantified HyperLTL formulas. We show that solving the enforcement problem for hyperproperties boils down to solving a variation of the synthesis or the satisfiability problem.

1.4 Deep Learning for Temporal Logics

Deep learning is a branch of machine learning, which itself is a branch of the large field of artificial intelligence. In deep learning, the goal is to train a neural network, consisting of multiple layers, on a specific task. A neural network can be trained in a supervised fashion, where labeled examples are provided to the network during the training phase, in a semi-supervised fashion, where only a tiny fraction of the data is labeled, or unsupervised, where there is no labeled data at all. Deep learning has become the state-of-the-art for many human-like tasks, such as computer vision [107, 201], translation [222], or board games [155, 193]. However, there is still the perception that deep neural networks cannot solve complex logical tasks reliably. Therefore, deep learning for formal methods frameworks has focused on sub-problems within larger logical frameworks, such as computing heuristics in solvers [136, 12, 188] or predicting individual proof steps [146, 94, 13, 112]. As

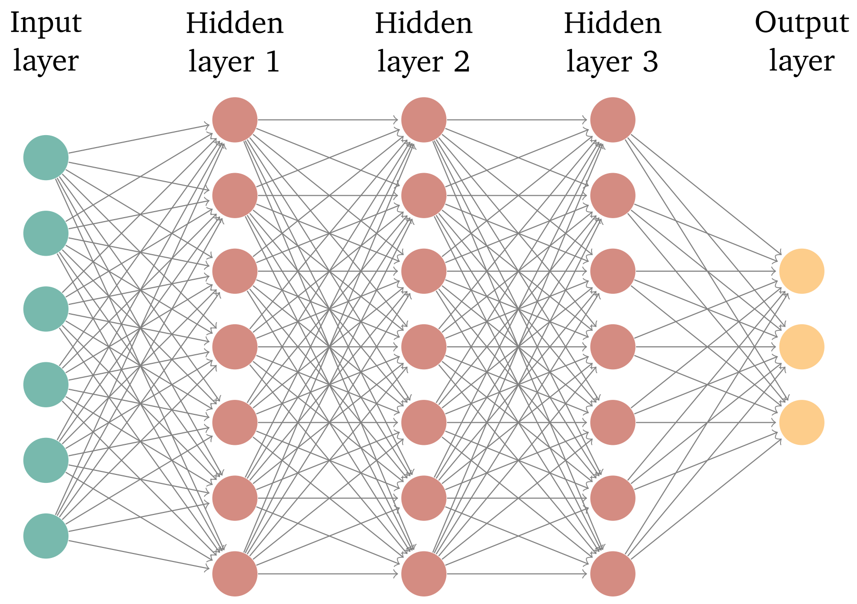


Figure 1.5: A sketch of a deep neural network with 6 input neurons, three hidden layers with 7 neurons each and an output layer with 3 neurons.

mentioned in Section 1.1, the assumption that deep learning is not yet ready to tackle hard logical questions, however, was drawn into question. We train a deep neural network on temporal logical reasoning tasks leading to verifiable predictions of the network. In the following, we provide an overview on Part II of this thesis. We give an overview on the challenges and our solutions for training a deep neural network on temporal logical reasoning tasks.

The architecture of a deep neural network [98] (see Figure 1.5) is inspired by the human brain. It consists of connected neurons, grouped into multiple *layers* (an input layer, multiple hidden layers, and an output layer), and can thus transmit signals to other neurons. Neurons and edges are labeled with weights that determine whether a neuron fires a signal. These weights are learned during a training process.[6] In a supervised learning setting (which we will focus on in this thesis), a deep neural network is trained by providing examples that contain a tuple of the input and the expected result. For example, in image classification for road signs, the inputs are pictures of road signs labeled with "Stop", "RightOf Way", and so on. Given a picture of a road sign, the expected output of the neural network is a correct prediction of its label. The neural network training was successful if it generalizes to examples (of the same distribution) that it has never seen during training.

In this thesis, we study the problem of applying deep learning to logical meth-

[6]Learning methods are typically implemented via a backpropagation algorithm [98].

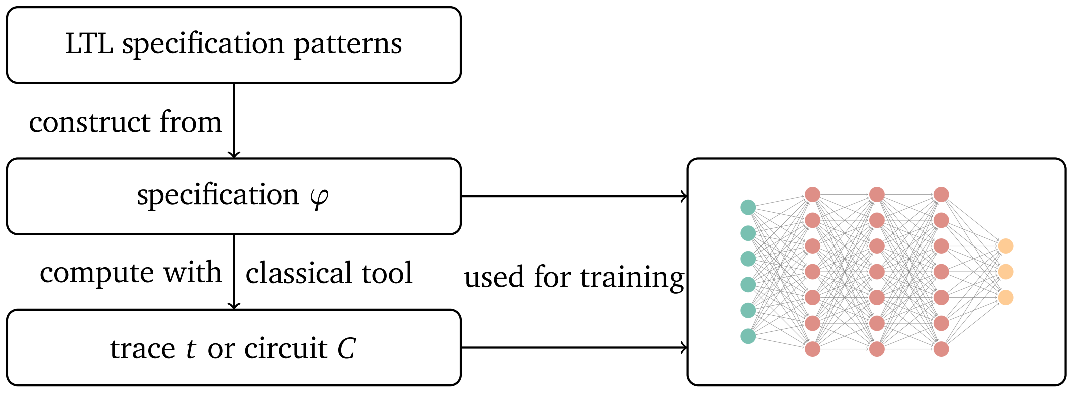


Figure 1.6: Overview of the neural network training process for logical methods of LTL. A specification is constructed by combining multiple specification patterns. Depending on the problem, we either construct a satisfying trace t (satisfiability) or a circuit implementation C (synthesis). The deep neural network is then trained on these data in a supervised fashion.

ods of linear-time temporal logic (LTL). We focus in this thesis on linear-time temporal logic (LTL) as the baseline for the machine learning experiments since LTL is the foundation of many specification languages (such as the industrial specification langauge and IEEE standard "property specification language" (PSL) [114] or HyperLTL) and widely used in the verification community. Logical methods of temporal logics are computationally challenging problems in general. LTL satisfiability is PSPACE-complete [196], and LTL synthesis is 2-EXPTIME-complete [166]. Computing solutions to the satisfiability or synthesis problem thus becomes unfeasible quickly when applying classical algorithms, such as automaton constructions.

The second research question, tackled in Part II of this thesis, is to provide a way to cope with these high computational costs. To this end, we provide the first approach to apply machine-learning, more specificially a Transformer, to problems of temporal logics *end-to-end*. The main benefit is that the temporal logic serves as a link between requirements in natural language and automatically verifiable predictions of the neural network. Successful training of a deep neural network on temporal logical problems would immediately yield powerful and fast heuristics. The computation time of predictions from a neural network can be expected to be a fraction of, for example, a classical synthesis algorithm that constructs an implementation via search. The main challenges of applying deep neural networks to logical methods for temporal logics end-to-end are the following: First, there exists no naturally occurring distribution of LTL formulas and their solutions (either traces or circuits), such that training data must be constructed synthetically; second, the network's output gives no guarantees whatsoever; and third, there are many (possibly infinite) solutions to a formula in temporal logic.

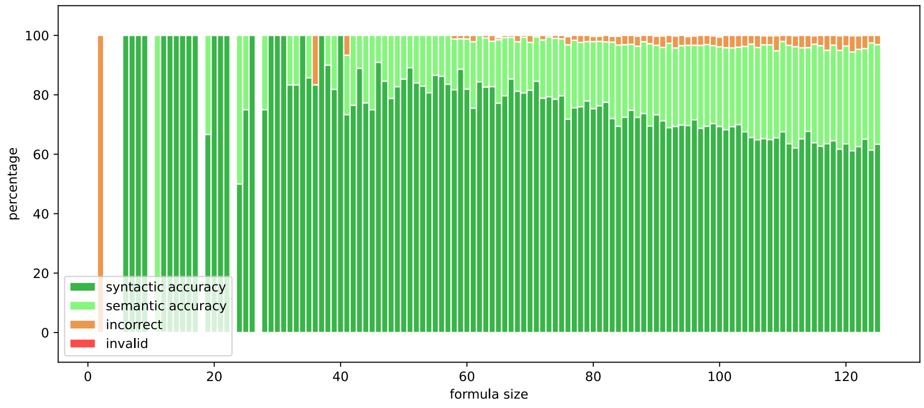


Figure 1.7: Performance of our best model trained on practical pattern formulas. The x-axis shows the formula size. Syntactic accuracy, i.e., where the deep neural network agrees with the classical tool are displayed in dark green. Instances where the deep neural network deviates from the classical tools output but still provides correct output are displayed in light green; incorrect predictions in orange.

We tackle the first problem by providing a data set constructed from specification patterns either provided by the literature [53] or mined from the annual synthesis competition SYNTCOMP [118]. For example, the following LTL formula is a request-response pattern that is widely used in specifying controller:

$$\square(request_i \rightarrow \diamondsuit grant_i) \ .$$

The formula states that at any point in time a request of a process i must be eventually granted. Combining this request pattern with the mutual exclusion pattern in Figure 1.1 is sufficient to specify a complete controller managing a shared resource. Figure 1.6 depicts an overview of our training process for a neural network on temporal logics presented in this thesis.

The second problem is that neural networks give no guarantees on their predictions. For logical methods of temporal logics, we can, however, make use of the following: verifying a solution is typically easier than coming up with a solution. The same holds for LTL. Finding a trace that satisfies an LTL formula is known to be PSPACE-complete [196], whereas verifying the trace is only in AC^1(logDCFL) [130]; and constructing a circuit is known to be 2-EXPTIME-complete [166], whereas model checking the resulting circuit is only in PSPACE [196]. We utilize this by verifying the solutions provided by the neural network with classical formal methods tools.

The last challenge is that there exist (possibly infinitely) many solutions to a temporal logical formula. We, thus, train on symbolic data that is as under-specified as possible. For example, instead of providing an explicit trace, we train the neural

(assumptions)
$\Box(\Diamond(\neg(i_0)))$
$\bigcirc(\Box((\neg(o_2))\vee(((\neg(i_4))\wedge(\neg(i_1)))$
$\mathrm{U}((\neg(i_4))\wedge(i_1)))))$
$\rightarrow$
(guarantees:)
$\Box((i_0)\rightarrow(\Diamond((\neg(i_0))\vee(o_4))))$
$\Box((i_2)\rightarrow(\Diamond(o_0)))$
$\Box((i_1)\rightarrow(\Diamond(o_0)))$
$(\Box(\Diamond(o_4)))\rightarrow(\Box((\Diamond(i_4))\wedge(\Diamond(i_1))$
$\Box((i_4)\rightarrow(\Diamond(o_3)))$
$\bigcirc(\Box((\neg(o_4))\vee(\neg(o_2))))$
$\Box((o_1)\rightarrow(\bigcirc((i_1)\mathcal{R}(((i_1)$
$\rightarrow(o_2))\wedge((\neg(i_1))\rightarrow(o_0))))))$
$\Box((\bigcirc(o_3))\rightarrow(i_3))$

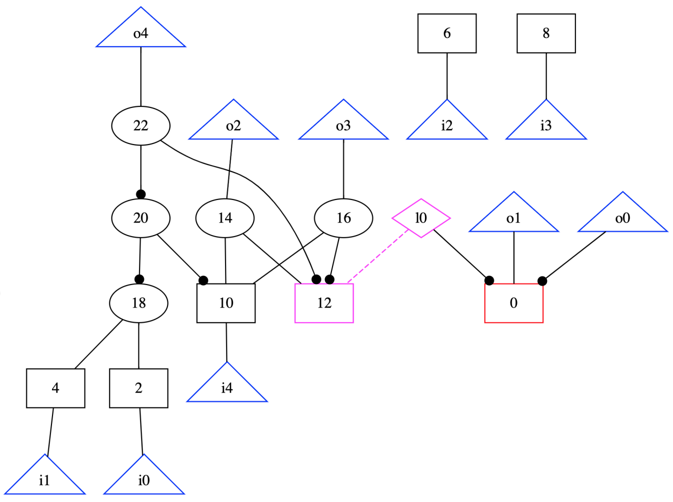


Figure 1.8: A specification in our test set, consisting of 2 assumption patterns and 8 guarantee patterns (left). A circuit, predicted by our deep learning model, satisfying the specification (right).

network on a symbolic trace, i.e., a sequence of propositional formulas that specify the possible valuations of the atomic propositions.

In Chapter 6, we demonstrate that deep neural networks can predict traces even for formulas from benchmarks from the literature on which the classical solver we used to generate the training data timed out; meaning they can generalize to larger and more complex formulas than seen during training. We show that the Transformer also generalizes to the semantics of the logics. While the models often deviate from the traces found by the classical solvers, they still predict correct traces to most formulas. Figure 1.7 visualizes the results of our best model on the trace generation task. We observe a significant large light green area, where the Transformer came up with different solutions to the formulas than the tool it was trained on. We furthermore provide out-of-distribution (OOD) tests that strengthen this observation.

In Chapter 7, we train a deep neural network on the LTL synthesis problem, i.e., the problem of constructing a circuit implementation directly out of an LTL specification. An annual competition is organized to track the improvement of algorithms and tools over time (SYNTCOMP) [118]. We ensure that the synthetic data we train on is sufficiently close to human-written specifications by mining common patterns from the specifications used in the synthesis competitions. We show that hierarchical Transformers trained on this synthetic data solve a significant portion of problems from the synthesis competitions and even OOD examples from a recent case study on smart homes [5]. Figure 1.8 depicts an example specification, constructed from mined specification patterns of the synthesis competition, of the held-out test set on which the neural network is evaluated. We are using a *hierarchical* Transformer [141], which can directly construct a circuit satisfying this specification. Overall, the best performing model achieved 81.25% accuracy on the held-out test set. We also tested the model on the SYNTCOMP benchmarks, where it solved 68.3% of the 145 instances within its space limits. Despite common belief, with the contributions of this thesis, direct machine learning approaches can be used to augment classical algorithms in verification tasks already today.

1.5 Publications

This thesis is based on the following publications.

[42] The Hierarchy of Hyperlogics. Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019). ©2019 IEEE.

[72] Realizing ω-regular Hyperproperties. Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Leander Tentrup. 32nd International Conference on Computer-Aided Verification (CAV 2020).

[43] Enforcing Hyperproperties. Norine Coenen, Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Yannick Schillo. 19th International Symposium on Automated Technology for Verification and Analysis (ATVA 2021).

[102] Teaching Temporal Logics to Neural Networks. Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus N. Rabe, and Bernd Finkbeiner. Ninth International Conference on Learning Representations (ICLR 2021).

[182] Neural Circuit Synthesis from Specification Patterns. Frederik Schmitt, Christopher Hahn, Markus N. Rabe, and Bernd Finkbeiner (NeurIPS 2021).

1.6 Contributions

In the first part of this thesis, we identify the theoretical borders for which the two core logical problems, i.e., the satisfiability and realizability problem, remain decidable when climbing up in the expressiveness hierarchy of hyperlogics. As a practical application, where both of these problems play an important role, we consider the enforcement of temporal hyperproperties for black box-systems.

In the second part of this thesis, we provide a data generation and a training method for solving the satisfiability and realizability problem of linear-time temporal logic (LTL) end-to-end with a deep neural network. This contribution paves the way for the development of more efficient heuristics and hybrid algorithms for temporal reasoning. In more detail, this thesis makes the following contributions.

1.6.1 Part I: Logical Methods for Temporal Reasoning

Logical methods for ω-regular temporal hyperproperties [42, 72]. In the realm of linear-time hyperlogics, we study the satisfiability and synthesis problem of HyperQPTL, a hyperlogic for ω-regular temporal hyperproperties. We can identify the exact borders of decidable fragments. These fragments even contain promptness properties. We provide encouraging experimental results that show that it is indeed possible to synthesize an arbiter respecting promptness from a HyperQPTL specification despite the high computational cost.

Logical methods for branching-time temporal hyperproperties [42]. We study the satisfiability problem of the branching-time hyperlogic HyperCTL*, for which we can identify decidable fragments. For the synthesis problem, we shortly remark that HyperCTL* inherits the decidable fragments of HyperLTL and is undecidable in general.

Enforcing temporal hyperproperties [43]. By elaborating the notion of a *sound* and *transparent* enforcement mechanisms for temporal hyperproperties, we contribute the necessary foundations for studying this problem algorithmically. We identify under which restrictions and in which trace input models [73, 74] enforcement mechanisms exist and provide efficient algorithms. The algorithmic solutions are based on solving either a variation of the synthesis or the satisfiability problem. We conduct experiments, showing that algorithms can perform well without significant overhead at runtime and can easily handle very long traces. The bottleneck is the initial solving of the synthesis problem.

1.6.2 Part II: Deep Learning Methods for Temporal Reasoning

Teaching temporal logics to neural networks [102]. We train a Transformer on the problem to predict a trace to a given LTL formula. We conduct various experiments, showing that deep neural networks generalize to the semantics of logics with our data generation method. While they often deviate from the solutions found by the classical solvers, with which we generated the training data, they still predict correct solutions to most formulas. We also provide experiments showing that models could solve instances from benchmarks from the literature on which the classical solver timed out. Furthermore, we performed experiments on propositional logic and out-of-distribution testing to confirm these observations. Overall, this contributes the first end-to-end training method of a deep neural network for a challenging logical task in verification.

Neural circuit synthesis from specification patterns [182]. We contribute a data generation and training method to train a hierarchical Transformer [141] to synthesize hardware circuits directly out of temporal logical specifications. We provide a mining method for LTL specification patterns from the annual synthesis competition SYNTCOMP [116]. We show that the machine learning models trained on this synthetic data solve many problems from the synthesis competitions. We also conduct experiments on out-of-distribution examples from a recent case study on secure smart homes [5]. This thesis provides a solution to the problem that applying machine learning to hardware synthesis suffers from a severe lack of sufficient training data.

1.7 Related Work

Temporal logics. Temporal logics have been studied in computer science since their introduction by Pnueli [165]. Since then, many extensions have been developed: e.g., computation tree logic (CTL/CTL*) [37, 57], signal temporal logic (STL) [148], or temporal logics for hyperproperties (HyperLTL) [40]. Logical methods for temporal logics have been studied extensively over the years, e.g., LTL satisfiability [139, 175, 185, 139, 138, 186] or LTL synthesis [83, 84, 22, 67, 152]. Other verification methods, such as model checking, or monitoring, have also been studied for LTL [39, 15, 85] and for various logics, e.g., STL monitoring [50] or CTL model checking [38].

Temporal hyperproperties. After the introduction of the linear-time hyperlogic HyperLTL and the branching-time hyperlogic HyperCTL* [40], formal methods for temporal hyperproperties have been studied extensively: satisfiability [70, 89, 149, 71, 75], model checking [82, 79, 111, 80], program repair [24], monitoring [77, 103, 23, 25, 198, 76, 78, 7], synthesis [73, 74], and expressiveness studies [42, 129, 88] have been conducted. There are various extensions of HyperLTL and HyperCTL*, e.g., HyperSTL [157], PHL [49], HyperQPTL [171], or HyperCTL$^*_{lp}$ [26].

Classic synthesis tools. The hardware synthesis problem traces back to the definition of the problem by Alonzo Church in 1957 [36], thus also called Church's Problem. With theoretical solutions, already in 1969 by Büchi and Landweber [31]. From a foundational point of view, advances have been made algorithmically, e.g., with a quasi-polynomial algorithm for parity games [33], conceptually with distributed [167] and bounded synthesis [84], or expressiveness-wise, e.g., GR(1) [164] synthesis, which is an efficient fragment of LTL or synthesis for security properties given in HyperLTL [74]. From a practical point of view, the field can build on a rich supply of tools (e.g. [22, 67, 152]). The first synthesis competition (SYNT-COMP) [118] was held in 2014, as part of the annual international conference on computer-aided verification (CAV).

Runtime Enforcement. Monitoring HyperLTL has been studied extensively (e.g.[77, 103, 23, 25, 198, 76, 78, 7]). Especially relevant is the work on realizability monitoring for LTL [55] using parity games. Existing work on runtime enforcement includes algorithms for safety properties [21, 220], real-time properties [174, 66], concurrent software specifications [147], and security policies [183, 59, 144]. For a tutorial on variants of runtime enforcement see [63]. Close related work is [156], which also studies the enforcement of general hyperproperties but independently of a concrete specification language (in contrast to this thesis). Systems are also assumed to be reactive and black-box, but there is no distinction between different trace input models. In this thesis, we mainly rely on parity game solving, while their enforcement mechanism executes several copies of the system to obtain executions that are related by the specification.

Property specification patterns. Dwyer et al. [53] identified 55 property specification patterns for temporal logics. Their general hierarchical specification pattern system can be mapped to temporal logics such as LTL and CTL. More patterns for temporal logical formulas are identified by [61, 110, 162]. In [127], the authors identified real-time specification patterns of real-time temporal logics, and in [99], a specification pattern system was presented for probabilistic properties formulated in probabilistic temporal logic.

Datasets for mathematical reasoning. Other works have studied datasets derived from automated theorem provers [19, 146, 94], interactive theorem provers [115, 121, 13, 112, 224, 168, 221, 140, 137, 205, 172, 159] (see [173] for a survey), symbolic mathematics [134], and mathematical problems in natural language [178, 181]. Close work to this thesis are the applications of Transformers to directly solve differential equations [134] and directly predict missing assumptions and types of formal mathematical statements [172]. We focus on a different problem domain, verification, and demonstrate that Transformers are roughly competitive with classical algorithms in that domain *on their dataset*. Learning has been applied to mathematics long before the rise of deep learning. Earlier works focused on ranking premises or clauses [32, 203, 204, 206, 151, 184, 122].

Neural architectures for logical reasoning. In [219], the authors present a reinforcement learning approach for interactive theorem proving. NeuroSAT [189] is a graph neural network [179, 143, 96, 223] for solving the propositional satisfiability problem. A simplified NeuroSAT architecture was trained for unsat-core predictions [188]. In [136], the authors have used graph neural networks on CNF to learn better heuristics for a 2QBF solver. In [62] the problem of logical entailment in propositional logic is studied using tree-RNNs. Entailment is a subproblem of satisfiability and the formulas considered in their dataset are much smaller than in this thesis. In [12], the authors applied graph neural networks to predict tactics for SMT solvers. In general, logical and mathematical reasoning with neural networks is a growing field of research. Stronger and flexible reasoning engines could serve as the basis for many applications, such as search, verification, synthesis and computer-aided design [199].

Language models applied to programs. Transformers have also been applied to programs for tasks such as summarizing code [68] or variable naming and misuse [108]. Other works have been focused on recurrent neural networks or graph neural networks for code analysis, e.g. [163, 101, 17, 217, 9]. Another area in the intersection of formal methods and machine learning is the verification of neural networks, e.g. [54, 191, 190, 194, 95, 113, 51].

Part I

Logical Methods for Temporal Reasoning

Chapter 2

Reactive Systems and Temporal Logics

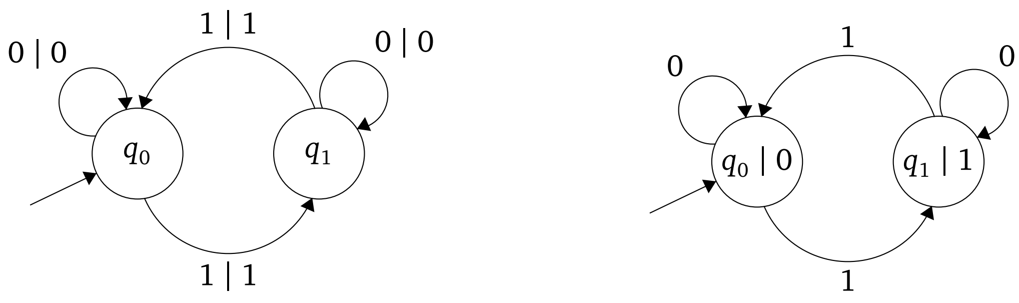
A reactive system continuously interacts with its environment. Examples are hardware circuits, communication protocols or embedded controllers [106]. They are indispensable and are deployed in safety-critical and security-critical systems. In order to *prove* that a reactive system adheres to a specification, both the specification and the model of a reactive system must be based on rigorous mathematics. In this chapter, we provide an overview of the mathematical models and constructions that the approaches in this thesis rely on.

We consider temporal logics as our formal specification framework for temporal reasoning. We differentiate between logics for trace properties, and hyperlogics[1] for hyperproperties. We begin by defining linear-time temporal logic (LTL), a logic for expressing trace properties, followed by HyperLTL, a hyperlogic for expressing hyperproperties. As a reference point, we define the notion of self-composition, which is a common algorithmic technique to verify *relations* between traces.

2.1 Reactive Systems

Reactive systems can be modeled as finite state machines, typically either as *Mealy* or *Moore* machines. The system models were first introduced back in 1955 [150] and 1956 [154], respectively. The next output of a Mealy machine is computed by its current state and the current input, whereas the next output of a Moore machine depends solely on the current state. Both system models are convertible into one another and many synthesis and model checking tools support both interpretations.

[1]Throughout this thesis, we will refer to logics that express temporal trace properties as temporal logics, and to logics that express temporal hyperproperties as (temporal) hyperlogics.



(a) Mealy machine. (b) Moore machine.

Figure 2.1: An example of a reactive system: a parity checker given as a Mealy and a Moore machine. The symbol | separates inputs (left) from outputs (right) for Mealy machines. For Moore machines, it separates the state name from the output.

Definition 1. *A Mealy machine is defined as a tuple $(S, s_0, \Sigma, O, \delta)$ with:*

- *S: a finite set of states,*
- *S_0: the initial state,*
- *Σ: the input alphabet,*
- *O: the output alphabet, and*
- *$\delta : S \times \Sigma \rightarrow S \times O$: a transition function.*

Definition 2. *A Moore machine is defined as a tuple $(S, s_0, \Sigma, O, \delta, G)$ with:*

- *S: a finite set of states,*
- *S_0: the initial state,*
- *Σ: the input alphabet,*
- *O: the output alphabet,*
- *$\delta : S \times \Sigma \rightarrow S$: a transition function, and*
- *$G : S \rightarrow O$: an output labeling.*

As an example of a simple reactive system, we consider a parity bit checker. A parity bit is a simple error detecting technique typically used in data transmission or communication [226]. When transmitting a bitstring, a single bit is added, which is 1 if the number of 1's in the bitstring is even and 0 if the number of 1's in the bitstring is odd. When submitting the 4-bitstring 1010, a parity bit 1 is added, resulting in 10101. If there is now a single transmission error, i.e., one of the bits were flipped during the transmission, the receive notices this error. For example, if the received bitstring is 11101, the number of 1's is even, meaning that one of the bits must be flipped. If an even number of errors occur, no error can be detected. Note that this technique can only detect errors, but can not correct any errors. Figure 2.1 depicts a parity checker encoded as a Moore and a Mealy machine.

2.2 Temporal Logics

Temporal Logics have been a longstanding paradigm for formalizing specifications of reactive systems. They are widely used in the verification community and are the basis for industrial specification langauges like the IEEE standard "property specification language" (PSL) [114]. In the following, we present LTL, a temporal logic for *trace properties*, i.e., specification that refer to a single system execution and HyperLTL, a temporal hyperlogic for *hyperproperties*, i.e., specifications over *relations* of system executions. Hyperproperties are especially considered in security-critical systems, since many security policies, such as information-flow, define relations over system executions.

2.2.1 Linear-time Temporal Logic

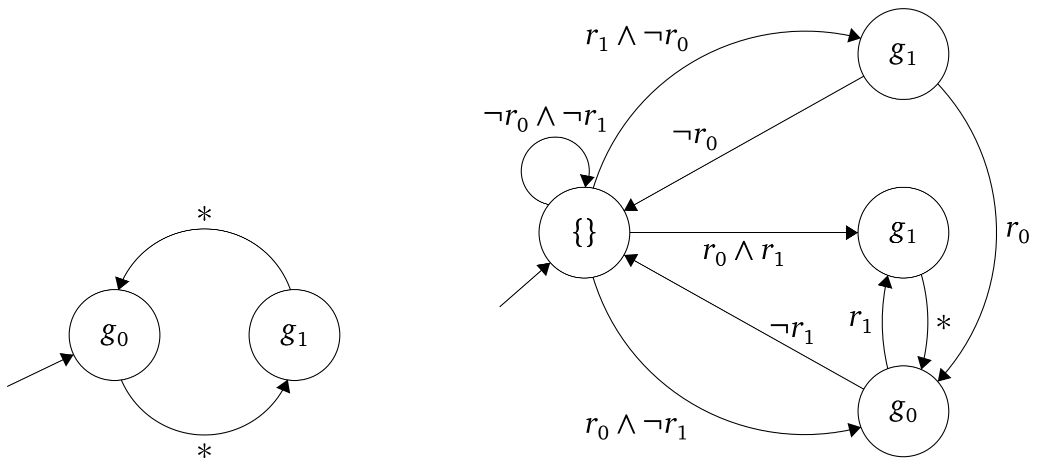
Linear-time temporal Logic (LTL) [165] was introduced by Amir Pnueli in 1977.

LTL Syntax. LTL combines the usual boolean connectives with temporal modalities such as the *Next* operator $\bigcirc$ and the *Until* operator $\mathcal{U}$. Formally, a *trace* t is an infinite sequence over subsets of atomic propositions AP. We define the set of traces $TR := (2^{AP})^\omega$. Let $p \in AP$ and $t \in TR$. The syntax of LTL is given by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi \ .$$

The formula $\bigcirc\varphi$ means that φ holds in the *next* position of a trace; $\varphi_1 \,\mathcal{U}\, \varphi_2$ means that φ_1 holds *until* φ_2 holds. We can derive several operators, such as $\diamondsuit\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$, $\square\varphi \equiv \neg\diamondsuit\neg\varphi$, $\varphi_1 \mathcal{R} \varphi_2 \equiv \neg(\neg\varphi_1 \,\mathcal{U}\, \neg\varphi_2)$, and $\varphi_1 \ \mathcal{W} \varphi_2 \equiv (\varphi_1 \,\mathcal{U}\, \varphi_2) \vee \square\varphi_1$. $\diamondsuit\varphi$ states that φ will *eventually* hold in the future and $\square\varphi$ states that φ holds *globally*, $\varphi_1 \mathcal{R} \varphi_2$ states that φ_1 *releases* φ_2, and $\mathcal{W}$ is the *weak* version of the *until* operator.

Consider, for example, a controller that manages a shared resource. The controller receives requests (r) and provides grants (g) to two processes. We identify



(a) A controller simply alternating between g_0 and g_1.

(b) A controller, without spurious grants.

Figure 2.2: Two controllers satisfying mutual exclusion with a symbolic edge notation: the symbol $*$ denotes that this edge can always be taken, and propositional formulas represent sets of edges. We omit set brackets and state names.

which request and grant belong to which process by indexing it with the corresponding process id (either 0 or 1). We can specify the behavior of a controller with LTL as follows:

$$(\Box(\neg(g_0 \land g_1))) \tag{2.1}$$
$$\land\ (\Box(r_0 \to (\Diamond g_0))) \tag{2.2}$$
$$\land\ (\Box(r_1 \to (\Diamond g_1)))\ . \tag{2.3}$$

The conjunct 2.1 states that at no point in time, the controller is allowed to give both grants g_0 and g_1 at the same time. Conjuncts 2.2 and 2.3 state that every request r_i is eventually followed by grant g_i (for $i \in \{1, 2\}$). Figure 2.2 depicts two implementations of a controller.

The semantics of LTL is defined over (execution) traces of a system in discrete time, which are, in a reactive context, typically assumed to be infinite sequences of valuations of system variables, called atomic propositions. A subset $T \subseteq TR$ is called a *trace property*. We use the following notation to manipulate traces: let $t \in TR$ be a trace and $i \in \mathbb{N}$ be a natural number. $t[i]$ denotes the i-th element of t. Therefore, $t[0]$ represents the starting element of the trace. Let $j \in \mathbb{N}$ and $j \geq i$. $t[i, j]$ denotes the sequence $t[i]\ t[i+1] \ldots t[j-1]\ t[j]$. $t[i, \infty]$ denotes the infinite suffix of t starting at position i. A trace, thus, represents the behavior of a single run through a system model.

LTL Semantics. Formally, the semantics of LTL is defined as follows:

$$
\begin{array}{lll}
t \models p & \text{iff} & p \in t[0] \\
t \models \neg\varphi & \text{iff} & t \not\models \varphi \\
t \models \varphi_1 \vee \varphi_2 & \text{iff} & t \models \varphi_1 \text{ or } t \models \varphi_2 \\
t \models \bigcirc\varphi & \text{iff} & t[1,\infty] \models \varphi \\
t \models \varphi_1 \,\mathrm{U}\, \varphi_2 & \text{iff} & \text{there exists } i \geq 0 : t[i,\infty] \models \varphi_2 \\
 & & \text{and for all } 0 \leq j < i \text{ we have } t[j,\infty] \models \varphi_1 \ .
\end{array}
$$

Assume, for example, the controller (a) in Figure 2.2. This simple implementation, in fact, satisfies the LTL specification in Equation 2.1- 2.3. However, the output of the system is independent of the input and provides spurious grants. Consider, for example, the following trace:

$$\{g_0\}\ \{g_1\}\ \{g_0\}\ \{r_0, g_1\}\ \{g_0\} \ldots$$

If we would like to specify that the controller provides no spurious grants, we have to be more precise by adding the following constraints:

$$\wedge\ (r_0 \,\mathcal{R}\, \neg g_0) \tag{2.4}$$

$$\wedge\ (r_1 \,\mathcal{R}\, \neg g_1)\ . \tag{2.5}$$

The conjuncts 2.4 and 2.5 state that no grants g_i are given until a request r_i releases this obligation. We also have to make sure that this obligation is reset after giving out a grant, which is more technical:

$$\wedge\ (\Box((g_0 \wedge (\bigcirc(\neg r_0 \wedge \neg g_0))) \rightarrow (\bigcirc(r_0 \,\mathcal{R}\, \neg g_0)))) \tag{2.6}$$

$$\wedge\ (\Box((g_1 \wedge (\bigcirc(\neg r_1 \wedge \neg g_1))) \rightarrow (\bigcirc(r_1 \,\mathcal{R}\, \neg g_1)))) \tag{2.7}$$

$$\wedge\ (\Box((g_0 \wedge \Box\neg r_0) \rightarrow \Diamond\neg g_0)) \tag{2.8}$$

$$\wedge\ (\Box((g_1 \wedge \Box\neg r_1) \rightarrow \Diamond\neg g_1))\ . \tag{2.9}$$

The controller (b) in Figure 2.2 satisfies the conjuncts 2.1-2.9. Instead of simply alternating between both grants, the implementation reacts accordingly to the requests r_0 and r_1, while still satisfying mutual exclusion.

LTL Satisfiability

In this section, we lay out an automata-theoretic approach to solving the satisfiability problem of LTL. Formally, the problem is defined as follows.

Definition 3 (LTL Satisfiability). *An LTL formula φ over atomic propositions AP is satisfiable if there exists a trace $t \in TR$ that satisfies φ.*

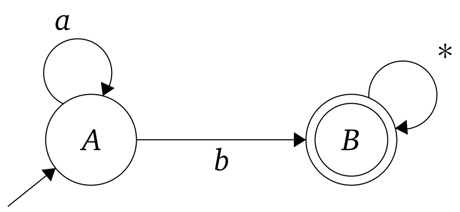


Figure 2.3: A Büchi automaton constructed from the LTL formula $a \,\mathrm{U}\, b$ with the online tool LTL2BA [92, 93]. The only accepting state is B, depicted as a double circle.

In 1962, Julius R. Büchi showed that a logical formula, given in monadic second-order logic of one successor (S1S), can be translated to an automaton over infinite words [29]. Since LTL is a syntactic fragment of quantified propositional logic (QPTL) (which is equally expressive to S1S [124]), this result also applies to LTL formulas. Note that QPTL, S1S and Büchi automata are equally expressive but more expressive than LTL. We define QPTL in detail in Chapter 3. In the following, we define ω-word automata and the Büchi acceptance condition.

Definition 4. *A (nondeterministic) ω-word automaton is defined as the following tuple $\mathcal{A} = (Q, Q_0, \Sigma, \delta, Acc)$:*

- *Q: a finite set of states,*
- *Q_0: the set of initial states,*
- *Σ: the input alphabet,*
- *$\delta \subset Q \times \Sigma \times Q$: the transition relation, and*
- *Acc: the accepting condition.*

We assume $\mathcal{A}$ to be *complete*, meaning that for every $q \in Q$ and $s \in \Sigma$, there exists a $q' \in Q$, s.t. $(q, s, q') \in \delta$. The automaton is called *determinstic* if δ is a function $\delta : Q \times \Sigma \to Q$, if Q_0 is a singleton set, and there is at most one $q' \in Q$, s.t. $(q, s, q') \in \delta$. The accepting condition Acc is defined over a *run* of the automaton. A run of $\mathcal{A}$ over an infinite input word $\sigma \in \Sigma^\omega$, is an infinite sequence of states $\tau \in Q^\omega$, s.t. the sequence starts with the initial state, i.e., $\tau[0] = q_0$ and the rest of the sequence follows the transitions, i.e., $(\tau[i], \sigma[i], \tau[i+1]) \in \delta$ for $i \in \mathbb{N}$. There are several acceptance conditions. Relevant for this thesis is the *safety* and the *Büchi* condition. The safety acceptance condition is given as a subset $Q_{safety} \subseteq Q$. A run of $\mathcal{A}$ is accepting if only safe states are visited. The Büchi condition [30] is necessary for the above-mentioned translation of S1S and LTL formulas into an automaton. It is a subset $Q_{\text{Büchi}} \subseteq Q$. A

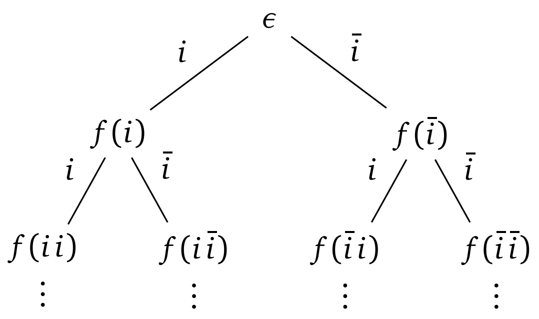


Figure 2.4: A strategy tree for the reactive realizability problem.

run of $\mathcal{A}$ is accepting if some state in $Q_{\text{Büchi}}$ is visited infinitely often. An example of a Büchi automaton is given in Figure 2.3. The automaton accepts all traces that satisfy the LTL formula $a \,\mathcal{U}\, b$, with the set of accepting states being $\{B\}$.

There exist multiple algorithms and tools for the translation of an LTL formula to a Büchi automaton. Most of them perform the translation in multiple steps with intermediate representations like generalized Büchi automata (GBA) [212] or alternating automata [207]. A solution to the satisfiability problem, i.e., either a trace or `unsat`, is computed by an emptiness check on the Büchi automaton, which can be performed on the fly.

Theorem 1 ([196]). *The satisfiability problem of LTL is PSPACE-complete.*

LTL Realizability

In this section, we give preliminaries on the realizability problem of LTL, which is the task of, given an LTL formula φ, to compute an implementation (e.g., a Moore or Mealy machine) that satisfies the specification. More precisely, the system is assumed to receive some inputs from an environment and has to react with outputs such that the specification is fulfilled. The realizability problem asks for the existence of a so-called *strategy tree*, where the edges are labeled with all possible inputs and the task is to find a function f that labels the nodes with the corresponding outputs. Figure 2.4 shows a strategy tree for a single input bit i.

Let disjoint sets of inputs and outputs $AP = I \mathbin{\dot\cup} O$ be given. Formally, a *strategy* $f : (2^I)^* \rightarrow 2^O$ maps sequences of input valuations 2^I to an output valuation 2^O. For an infinite word $w = w_0 w_1 w_2 \cdots \in (2^I)^\omega$, the trace corresponding to a strategy f is defined as $(f(\epsilon) \cup w_0)(f(w_0) \cup w_1)(f(w_0 w_1) \cup w_2) \ldots \in (2^{I \cup O})^\omega$.

Definition 5 (LTL Realizability). *An LTL formula φ over disjoint inputs and outputs $AP = I \mathbin{\dot\cup} O$ is realizable if there is a strategy $f : (2^I)^* \rightarrow 2^O$ that satisfies φ.*

Such a strategy can be computed by playing a 2-player parity game where the “system”-player, who tries to satisfy the specification, faces the “environment”-player, who tries to violate the specification. Intuitively, the environment-player has control over the inputs and the system-player must react with certain outputs. For a given LTL formula, we first construct a nondeterministic Büchi automaton in exponential time. This Büchi automaton can be translated in exponential time into a *deterministic* parity automaton.[2] Formally, a parity automaton is a generalization of a Büchi automaton: every $q \in Q$ is labeled with some natural number k and the parity accepting condition requires the smallest number that occurs infinitely often in a run to be even (or odd). The parity game is then played in an arena, which is constructed from a deterministic parity automaton.

Formally, a *parity game* $\mathcal{G}$ is a two-player game $(V_0, V_1, \Sigma_0, \Sigma_1, E_0, E_1, v_{in}, c)$, where V_0 and V_1 are the states belonging to player P_0 and P_1, respectively, Σ_0 and Σ_1 are the sets of actions, $E_0 : V_0 \times \Sigma_0 \to V_1$ and $E_1 : V_1 \times \Sigma_1 \to V_0$ are the transition functions, $v_{in} \in V_0$ is the initial state, and $c : V_0 \dot\cup V_1 \to \mathbb{N}$ is the coloring function. States belonging to P_0 and P_1 are required to alternate along every path in a parity game. For an infinite decision sequence $w = w_0^0 w_0^1 w_1^0 w_1^1 \ldots$, where the two players choose one of their possible actions in every step (i.e., $\forall i \in \mathbb{N}. w_i^0 \in \Sigma_0$ and $w_i^1 \in \Sigma_1$), the game generates an infinite play $r = v_0^0 v_0^1 v_1^0 v_1^1 \ldots$, where v_0^0 is the initial state and the edge function is followed according to w, i.e., $\forall i \in \mathbb{N}. p \in \{0,1\}. v_{i+p}^{1-p} = E_p(v_i^p, w_i^p)$. The play r is winning for player P_0 if and only if the highest color occurring infinitely often in the sequence $c(v_0^0)c(v_0^1)c(v_1^0)c(v_1^1)...$ is even. Otherwise, player P_1 wins. A strategy maps the history of the play to the next move that should be executed by the player whose turn it is. The strategy for player P_0 is a function $\sigma_0 : (\Sigma_0 \times \Sigma_1)^* \to \Sigma_0$, the strategy for player P_1 is a function $\sigma_1 : (\Sigma_0 \times \Sigma_1)^* \times \Sigma_0 \to \Sigma_1$. A decision sequence w is conforming to a strategy σ_p for $p \in \{0,1\}$, if $\forall i \in \mathbb{N}$ we have that $w_i^p = \sigma_p(w_0^0 w_0^1 ... w_{i+p-1}^{1-p})$. A strategy σ_p is called *winning* for player P_p if all decision sequences w conforming to σ_p generate plays that are winning for player P_p. There always exists a winning strategy for exactly one of the players since parity games are *determined* [58]. A state $v \in V_0 \dot\cup V_1$ is called winning for player P_p if this player has a winning strategy in the modified game $\mathcal{G}^v$ where the initial state v_{in} is set to v. The set of all states that are winning for player P_p is called the *winning region* for player P_p. Parity games are even *memoryless determined*, i.e., if player P_p has a winning strategy in state v, then P_p has a *positional strategy* $\sigma : V_p \to \Sigma_p$, such that for any decision sequence conforming to σ, the play starting in v is winning for p. Positional strategies give a decision for each of the player's states, independently of the history of the play. The exact complexity of parity game solving, i.e., determining the winning region, is still unknown. Current state-of-the-art algorithms perform in quasi-polynomial time [33, 161].

Theorem 2 ([166]). *The realizability problem of LTL is 2-EXPTIME-complete.*

[2]The computational cost comes from determinization of the automaton [177].

We use the terms realizability problem and synthesis problem interchangeably throughout this thesis.

2.2.2 HyperLTL

Security policies, such as noninterference [97] are *hyperproperties* [41], since they relate multiple computation traces. Temporal logics that only reason over single executions of a system are thus not capable of expressing them. In this section, we present a recently introduced extension of LTL, called HyperLTL[40], which is capable of expressing temporal hyperproperties. HyperLTL extends LTL with trace variables and trace quantifiers. With this mechanism, it is possible to express relations between traces. For example, one of the simplest hyperproperties is plain trace equivalence, i.e., every two traces must agree on all atomic propositions. The following HyperLTL formula, for example, expresses that all traces of a controller agree on the grants:

$$\begin{aligned}\forall \pi . \forall \pi' . \square (&{g_0}_\pi \leftrightarrow {g_0}_{\pi'}) \\ \land \square (&{g_1}_\pi \leftrightarrow {g_1}_{\pi'}) \ .\end{aligned}$$

The formula reads as "all traces π and π' must agree on grants g_0 and g_1 at every point in time". Controller (a) in Figure 2.2, in fact, adheres to this specification. Controller (b), however, does not satisfy this specification, since the grants provided depend on the input requests.

HyperLTL Syntax. Let $\mathcal{V}$ be an infinite supply of trace variables. Formally, the syntax of HyperLTL is given by the following grammar:

$$\begin{aligned}\psi &::= \exists \pi.\, \psi \mid \forall \pi.\, \psi \mid \varphi \\ \varphi &::= a_\pi \mid \neg \varphi \mid \varphi \lor \varphi \mid \bigcirc \varphi \mid \varphi \,\mathrm{U}\, \varphi\end{aligned}$$

where $a \in AP$ is an atomic proposition and $\pi \in \mathcal{V}$ is a trace variable. Note that atomic propositions are indexed by trace variables. The quantification over traces makes it possible to express properties like "on all traces ψ must hold", which is expressed by $\forall \pi.\ \psi$. Dually, one can express that "there exists a trace such that ψ holds", which is denoted by $\exists \pi.\ \psi$. The derived operators $\diamondsuit$, $\square$, $\mathcal{R}$, and W are defined as for LTL.

A more practical example is observational determinism [225], which is an information-flow policy. The specification requires that two traces, with the same observable input, appear the same to an observer, i.e., have the same observable outputs. The hyperproperty is formalized as follows:

$$\forall \pi . \forall \pi' . \square (\bigwedge_{i \in I} i_\pi \leftrightarrow i_{\pi'}) \rightarrow \square (\bigwedge_{o \in O} o_\pi \leftrightarrow o_{\pi'}) \ ,$$

where I denotes the observable inputs and O denotes the observable outputs[3]. The

[3]Throughout this thesis, we also use the following notation: $O_\pi \leftrightarrow O_{\pi'}$.

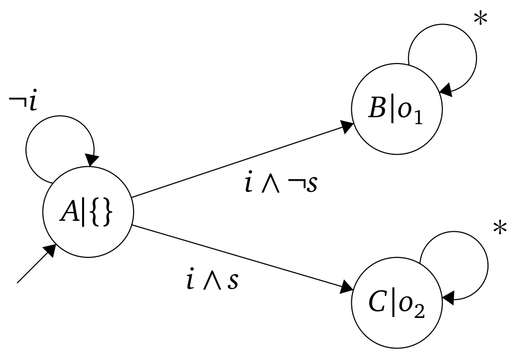


Figure 2.5: An example system with observable input i that leaks a secret s through observable outputs o_1 and o_2.

formula reads as "all traces π and π' must agree on every output $o \in O$ if the traces agree on ever input $i \in I$". Figure 2.5 depicts an example system that leaks the secret s through the observable outputs o_1 and o_2.

HyperLTL Semantics. Formally, a HyperLTL formula defines a *hyperproperty H*, which is a set of sets of traces, i.e., $H \in 2^{TR}$. A set T of traces satisfies the hyperproperty H if it is an element of this set of sets, i.e., if $T \in H$. The semantics of HyperLTL is given with respect to a *trace assignment* Π from $\mathcal{V}$ to *TR*, i.e., a partial function mapping trace variables to actual traces. The notation $\Pi[\pi \mapsto t]$ denotes that π is mapped to t, with everything else mapped according to Π and $\Pi[i,\infty]$ denotes the trace assignment that is equal to $\Pi(\pi)[i,\infty]$ for all π:

$$\begin{array}{lll}
\Pi \models_T \exists \pi.\psi & \text{iff} & \text{there exists } t \in T \; : \; \Pi[\pi \mapsto t] \models_T \psi \\
\Pi \models_T \forall \pi.\psi & \text{iff} & \text{for all } t \in T \; : \; \Pi[\pi \mapsto t] \models_T \psi \\
\Pi \models_T a_\pi & \text{iff} & a \in \Pi(\pi)[0] \\
\Pi \models_T \neg\psi & \text{iff} & \Pi \not\models_T \psi \\
\Pi \models_T \psi_1 \vee \psi_2 & \text{iff} & \Pi \models_T \psi_1 \text{ or } \Pi \models_T \psi_2 \\
\Pi \models_T \bigcirc \psi & \text{iff} & \Pi[1,\infty] \models_T \psi \\
\Pi \models_T \psi_1 \mathcal{U} \psi_2 & \text{iff} & \text{there exists } i \geq 0 : \Pi[i,\infty] \models_T \psi_2 \\
 & & \text{and for all } 0 \leq j < i \text{ we have } \Pi[j,\infty] \models_T \psi_1 \; .
\end{array}$$

Self-composition

A common technique to verify hyperproperties is to first compute the *self-composition* of a system [14]. With this technique the verification problem for hyperproperties can be reduced to the verification problem of a trace property [82]. For computing

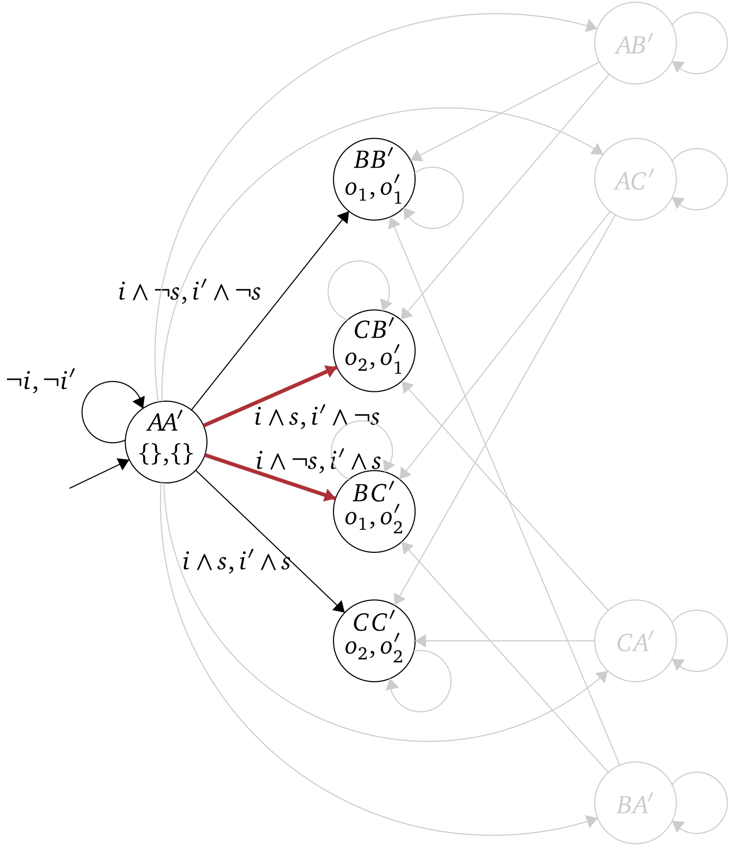


Figure 2.6: Self-composition of the example system in Figure 2.5. The second copy of the system is indicated by a $'$ symbol. Parts that already violate the premise of the specification are grayed out. Some edge labels are omitted for readability.

the self-composition, first the system is copied and then composed into a product system simulating both runs of the system in parallel. The self-composition of the example system leaking a secret (see Figure 2.5) is visualized in Figure 2.6. The following LTL formula specifies observational determinism for the self composition, which is a trace property:

$$(\square(i \leftrightarrow i')) \rightarrow \bigwedge_{i \in \{1,2\}} (\square(o_i \leftrightarrow o'_i)) \ .$$

The parts of the self-composition where traces differ in their observable inputs i are grayed out, since they violate the premise of observational determinism already. Problematic traces are the ones that agree on their input, as observational determinism requires that they should also agree on their observable output. Traces depicted in red, however, violate this specification. An attacker could infer the secret s be repeatedly observing multiple runs of the system.

Although motivated from security-critical applications, note that there are many more application areas for hyperproperties. For example, other hyperproperties are, as mentioned in the introduction, robustness [200], distributivity [167], symmetry [82], or fault-tolerance [131].

HyperLTL Satisfiability

In contrast to logics for trace properties where the satisfiability problem asks for a satisfying trace, the satisfiability problem of hyperlogics asks for a non-empty *set* of traces. Formally, the problem is defined as follows.

Definition 6 (HyperLTL Satisfiability)**.** *A HyperLTL formula φ is satisfiable if there exists a non-empty set of traces T such that $\Pi \models_T \psi$, where Π is the empty trace assignment.*

If it is clear from the context, we omit Π and simply write $T \models \psi$. We call T a model of ψ. Constructing a set of traces poses a new challenge and, in fact, leads to undecidability of the logic in general [70]. However, the problem remains decidable for a large class of hyperproperties, such as information-flow policies that are expressible in the $\forall^*$ fragment. We arrange the findings on this matter on the quantifier structure of the HyperLTL formula. An overview is depicted in Table 2.1.
The satisfiability problem has been shown to be undecidable for the full logic by a reduction from Post's Correspondence Problem (PCP) [70]. This result was then refined by a reduction from the recurring tiling problem, showing that the problem is Σ_1^1-complete.

Theorem 3 ([89])**.** *The satisfiability problem of HyperLTL is Σ_1^1-complete.*

Excursus: Restricting Models or Temporal Depth

In this excursus, we will have a quick look on further restrictions on the satisfiability problem of HyperLTL investigated in the literature. In this thesis, however, we will focus on structuring fragments based on the quantifier structure.

Fragment of HyperLTL	Complexity
$\forall^*$	PSPACE-complete [70]
$\exists^*$	PSPACE-complete [70]
$\exists^*\forall^*$	EXPSPACE-complete [70]
Full Logic	Σ_1^1-complete [89]

Table 2.1: Complexity of the satisfiability problem of HyperLTL by quantifier structure.

It is often sufficient to reason about a fixed number of traces, e.g., two in the case of information-flow policies. When restricting the number of universal quantifiers in a formula to a bound $b \in \mathbb{N}$, it can be translated to an equisatisfiable LTL formula of polynomial size.

Corollary 1 ([70]). *The satisfiability problem of HyperLTL is PSPACE-complete for the bounded $\exists^*\forall^*$ fragment.*

Similarly, instead of restricting the number of quantifiers in the formula, we can restrict the cardinality of the trace models.

Theorem 4 ([149]). *The following problem is EXPSPACE-complete: Given a HyperLTL formula φ and $k \in \mathbb{N}$ does φ have a model with at most k traces?*

Another dimension to analyze the complexity of HyperLTL formulas is their nesting of temporal operators [149]. The authors showed that the lower bounds remain the same for a temporal depth of two or higher. For a temporal depth of one, however, the complexity of the problem decreases. Table 2.2 summarizes their results.

HyperLTL Realizability

It has been shown that the HyperLTL realizability problem subsumes many other studied variants of the realizability problem studied in the literature [74], such as synthesis under incomplete information [133], distributed synthesis [167], asynchronous distributed synthesis [180], symmetric synthesis [56], or fault-tolerant synthesis [86, 87]. The problem, however, is computationally very challenging and becomes undecidable quickly. Table 2.3 summarizes the known results.

The problem is formally defined as follows. For any trace $w = w_0 w_1 w_2 \ldots \in (2^{I \cup O})^\omega$ and strategy $f : (2^I)^* \rightarrow 2^O$, we lift the set containment operator $\in$ defining that $w \in f$ iff $f(\epsilon) = w_0 \cap O$ and $f((w_0 \cap I) \cdots (w_i \cap I)) = w_{i+1} \cap O$ for all $i \geq 0$. We say that a strategy f satisfies a HyperLTL formula φ over $\mathrm{AP} = I \mathbin{\dot\cup} O$ iff $\{w \mid w \in f\}$ satisfies φ.

Fragment of HyperLTL	Temporal Depth 1
$\exists^*$ / $\forall^*$	NP-complete
$\exists^*\forall^*$	NEXPTIME-complete
$\exists^*\forall\exists^*$	in N2EXPTIME*
$\forall^2\exists^*$	undecidable

Table 2.2: Results on restricting the temporal depth of a HyperLTL formula, where $*$ denotes that the lower bound only holds for U-free formulas [149].

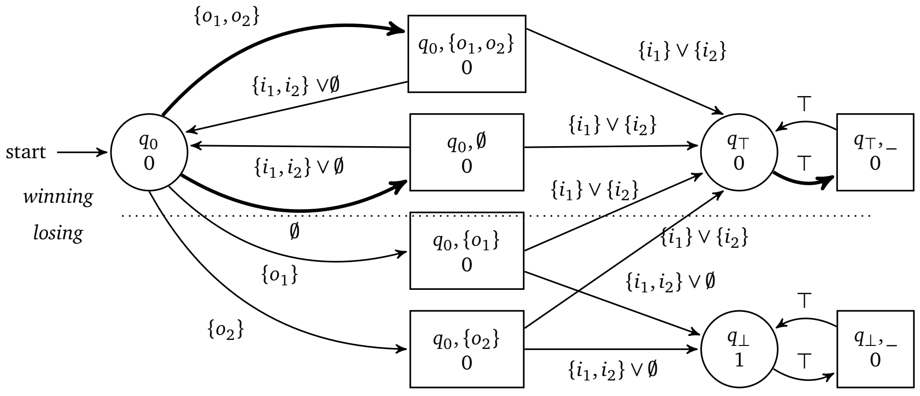


Figure 2.7: Parity game for observational determinism. States of the system player P_0 are depicted as circles.

Definition 7 (HyperLTL Realizability). *A HyperLTL formula φ over disjoint inputs and outputs $AP = I \dot\cup O$ is realizable if there is a strategy $f : (2^I)^* \to 2^O$ that satisfies φ.*

For example Figure 2.7 depicts a parity game for observational determinism. Let $\Sigma_0 = \{o_1, o_2\}$ be the set of low-security outputs and $\Sigma_1 = \{i_1, i_2\}$ be the set of low-security inputs. As long as i_1 and i_2 agree, o_1 and o_2 also have to agree. In this simplified example, a winning strategy would be to always choose o_1 equivalent to o_2, independently of the inputs. The system player P_0 has a winning strategy from the upper states using only bold transitions from system states.

Intuitively, the realizability problem of HyperLTL becomes undecidable quickly, because the $\forall^*$ fragment of HyperLTL can express *(in)dependence* [74]. This is a hyperproperty, which enables the encoding of the *distributed* realizability problem of LTL, which is known to be undecidable [167]. We will discuss the realizability problem of temporal hyperproperties in more detail in the next chapter, when considering ω-regular hyperproperties.

Fragment of HyperLTL	Complexity
$\exists^*$	PSPACE-complete
$\forall^*$	Undecidable
linear $\forall^*$	NONELEMENTARY
$\exists^*\forall^1$	3-EXPTIME
$\exists^*\forall^{>1}$	Undecidable
$\forall^*\exists^*$	Undecidable

Table 2.3: Decidability results for the realizability problem of HyperLTL [74].

Chapter 3

Logical Methods for ω-regular Temporal Hyperproperties

Hyperlogics can not only specify functional correctness but may also enforce the absence of information leaks or the presence of information propagation. There is a great practical interest in information flow control, which makes synthesizing implementations that satisfy hyperproperties highly desirable. As mentioned in the last chapter, it was shown that the synthesis problem of HyperLTL, although undecidable in general, remains decidable for many fragments, such as the $\exists^*\forall$ fragment. Furthermore, a *bounded synthesis* procedure was constructed, for which a prototype implementation based on BoSy[67, 73, 44] showed promising results.

HyperLTL is, however, intrinsically limited in expressiveness. For example, promptness is not expressible in HyperLTL. *Promptness* is a property stating that there is a bound b, common for all traces, on the number of steps up to which an event e must have happened. Additionally, just like LTL, HyperLTL can express neither ω-regular nor epistemic properties [171, 26]. We will, thus, move up in the expressiveness hierarchy of hyperlogics [42] to HyperQPTL. We will determine the theoretical borders for which the satisfiability and realizability problem remain decidable.

HyperQPTL subsumes epistemic extensions of temporal logics such as LTL_K [104], as well as the first-order hyperlogic FO$[<, E]$ [88, 171, 42]. Epistemic properties are statements about the transfer of knowledge between several components. The *dining cryptographers problem* [35] describes an exemplary epistemic specification. Three cryptographers sit at a table in a restaurant. Either one of the cryptographers or the NSA must pay for their meal. The question is whether there is a protocol where each cryptographer can determine whether the NSA or one of the cryptographers paid the bill without revealing the identity of the paying cryptographer. Its expressiveness makes HyperQPTL particularly interesting.

HyperQPTL extends HyperLTL with quantification over sequences of new propositions. What makes the logic particularly expressive is that the trace quantifiers and propositional quantifiers can be freely interleaved. With this mechanism, HyperQPTL can not only express all ω-regular properties over a sequence of n-tuples; it truly interweaves trace quantification and ω-regularity. For example, promptness can be stated as the following HyperQPTL formula:

$$\exists b. \forall \pi. \ \Diamond b \land (\neg b \ \mathcal{U} e_\pi) \ .$$

The formula states that there exists a sequence $s \in (2^{\{q\}})^\omega$, such that event e holds on all traces before the first occurrence of b in s.

We show that satisfiability is decidable for large fragments of the logic. The decidable HyperQPTL fragments can be described in terms of their quantifier prefix. We show that fragments consisting of a propositional $\forall\exists$ quantifier alternation followed by one type of trace quantifier remain decidable. No $\forall\exists$ propositional quantifier alternation is allowed, however, if followed by both types of trace quantification (even $\exists^*\forall^*$).

Propositional quantification also has an impact on the realizability problem: it becomes undecidable when combining a propositional $\forall\exists$ quantifier alternation with a single universal trace quantifier. However, we show that the synthesis problem of large HyperQPTL fragments remains decidable, where one of these fragments contains promptness properties. We partially obtain these results by reducing the HyperQPTL realizability problem to the HyperLTL realizability problem. We extend the BoSy bounded synthesis tool to synthesize systems respecting HyperQPTL specifications based on this reduction. We provide promising experimental results of our prototype implementation: using BoSy and HyperQPTL specifications, we synthesized arbiters that respect promptness.

Results in this chapter are based on the satisfiability part in “The Hierarchy of Hyperlogics” [42], which was joint work with Norine Coenen, Bernd Finkbeiner, and Jana Hofmann and the realizability part in “Realizing ω-regular Hyperproperties” [72], which was joint work with Bernd Finkbeiner, Jana Hofmann, and Leander Tentrup. Furthermore, the author would like to thank Jonni Virtema for pointing out inconsistencies. The chapter is structured as follows. We first define HyperQPTL formally in Section 3.1 and give an intuition on ω-regularity. We then proceed by considering the satisfiability and realizability problem of HyperQPTL in Section 3.2 and Section 3.3 respectively. Lastly, Section 3.4 presents experiments for the prototype implementation of a bounded synthesis algorithm for HyperQPTL.

3.1 HyperQPTL

LTL, and thus HyperLTL, is limited in expressiveness: counting languages, such as $\mathcal{L} = (\emptyset\emptyset)^*\{h\}^\omega$, can not be expressed. Quantified Propositional Temporal Logic

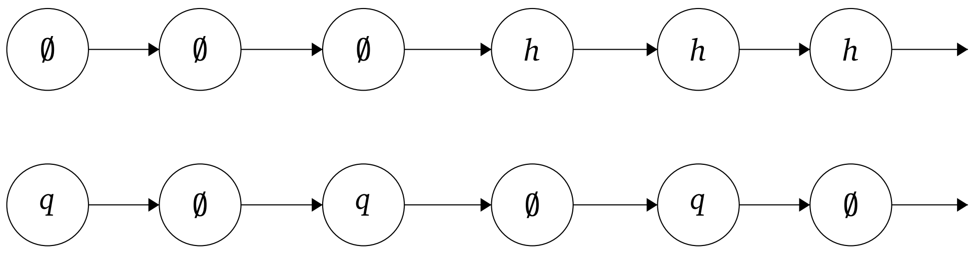


Figure 3.1: Quantification mechanism of QPTL for an example system trace (top). A new trace in $(2^{\{q\}})^\omega$ (below) is constructed, that marks every odd position.

(QPTL) can express such ω-regular trace properties with a mechanism of quantifying over propositions. As mentioned in Chapter 2, note that LTL is a syntactic fragment of QPTL. Additionally QPTL is exactly as expressive as S1S and their definable languages are thus Büchi recognizable.

QPTL Syntax. QPTL [195] extends Linear Temporal Logic (LTL) with quantification over propositions. QPTL formulas φ are defined as follows:

$$\begin{aligned} \varphi &::= \exists q.\, \varphi \mid \forall q.\, \varphi \mid \psi \\ \psi &::= q \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc\psi \mid \Diamond\psi \ , \end{aligned}$$

where $q \in \mathrm{AP}$ and AP is a set of atomic propositions. For simplicity, we assume that variable names in formulas are cleared of double occurrences. The example language above, i.e., $\mathcal{L} = (\emptyset\emptyset)^*\{h\}^\omega$, specifies that a halting symbol h may only appear after an even number of steps. This language is expressible in QPTL as follows:

$$\exists q.(q \wedge \Box(q \leftrightarrow \neg\bigcirc q) \tag{3.1}$$
$$\wedge \Box(\bigcirc h \rightarrow (h \vee q)) \tag{3.2}$$
$$\wedge \Diamond h \wedge \Box(h \rightarrow \bigcirc h) \ . \tag{3.3}$$

Figure 3.1 visualizes the quantification mechanism of QPTL. The Conjunct 3.1 ensures that every odd position of a new non-system trace in $(2^{\{q\}})^\omega$ is marked by a q. The Conjunct 3.2 states that the first h is only at a non-marked position. The last conjunct then requires that h eventually appears (and stays) on the system trace.

For defining the semantics of QPTL, we use the following notation. Given a trace $t \in (2^{AP})^\omega$ and a trace $t_q \in (2^{\{q\}})^\omega$, we define $t[q \mapsto t_q]$ as the replacement of the occurrences of q in t according to t_q, such that $t[q \mapsto t_q] =_{\{q\}} t_q$ and $t[q \mapsto t_q] =_{\mathrm{AP}\setminus\{q\}} t$. We also lift this notation to sets of traces as $T[q \mapsto t_q] = \{t[q \mapsto t_q] \mid t \in T\}$.

QPTL Semantics. The semantics of φ over AP is defined with respect to a trace $t \in TR$:

$$
\begin{array}{lll}
t \models q & \text{iff} & q \in t[0] \\
t \models \neg\psi & \text{iff} & t \not\models \psi \\
t \models \psi_1 \vee \psi_2 & \text{iff} & t \models \psi_1 \text{ or } t \models \psi_2 \\
t \models \bigcirc\psi & \text{iff} & t[1,\infty] \models \psi \\
t \models \Diamond\psi & \text{iff} & \exists i \geq 0.\ t[i,\infty] \models \psi \\
t \models \exists q.\, \varphi & \text{iff} & \exists t_q \in (2^{\{q\}})^\omega.\ t[q \mapsto t_q] \models \varphi \\
t \models \forall q.\, \varphi & \text{iff} & \forall t_q \in (2^{\{q\}})^\omega.\ t[q \mapsto t_q] \models \varphi \ .
\end{array}
$$

We did not define the until operator U as native part of the logic. It can be derived using propositional quantification [119]. The boolean connectives $\wedge, \rightarrow, \leftrightarrow$, and the temporal operators globally $\Box$ and release $\mathcal{R}$ are derived as in Chapter 2.

Definition 8 (QPTL Satisfiability). *A QPTL formula φ is satisfiable if there exists a trace $t \in TR$ that satisfies φ.*

Due to the arbitrary quantifier alternations, the complexity of the satisfiability problem for QPTL is non-elementary in the number of alternations.

Theorem 5 ([197]). *The satisfiability problem of QPTL is decidable with NONELEMENTARY complexity.*

Similar to the lifting of LTL to HyperLTL, we can lift QPTL to HyperQPTL. Note that we allow an arbitrary interleaving of propositional and trace quantification.

HyperQPTL Syntax. Given a set AP of atomic propositions and a set $\mathcal{V}$ of trace variables, the syntax of HyperQPTL is defined as follows:

$$
\begin{aligned}
\varphi &::= \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \forall q.\, \varphi \mid \exists q.\, \varphi \mid \psi \\
\psi &::= a_\pi \mid q \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc\psi \mid \Diamond\psi \ ,
\end{aligned}
$$

where $a, q \in$ AP and $\pi \in \mathcal{V}$. As for QPTL, we assume that formulas are cleared of double occurrences of variable names. We require that in well-defined HyperQPTL formulas, each a_π is in the scope of a trace quantifier binding π, and each q is in the scope of a propositional quantifier binding q. Note that atomic propositions a_π refer to a quantified trace π, whereas quantified propositional variables q are independent of the traces.

HyperQPTL Semantics. The semantics of a well-defined HyperQPTL formula over AP is defined with respect to a set of traces $T \subseteq (2^{AP})^\omega$ and an assignment function $\Pi : \mathcal{V} \to T$. We define the satisfaction relation $\Pi, i \models_T \varphi$ as follows:

$$
\begin{array}{lll}
\Pi, i \models_T a_\pi & \text{iff} & a \in \Pi(\pi)[i] \\
\Pi, i \models_T q & \text{iff} & \forall t \in T.\, q \in t[i] \\
\Pi, i \models_T \neg\psi & \text{iff} & \Pi, i \not\models_T \psi \\
\Pi, i \models_T \psi_1 \vee \psi_2 & \text{iff} & \Pi, i \models_T \psi_1 \vee \Pi, i \models_T \psi_2 \\
\Pi, i \models_T \bigcirc\psi & \text{iff} & \Pi, i+1 \models_T \psi \\
\Pi, i \models_T \Diamond\psi & \text{iff} & \exists j \geq i.\ \Pi. j \models_T \psi \\
\Pi, i \models_T \exists\pi.\,\varphi & \text{iff} & \exists t \in T.\, \Pi[\pi \mapsto t], i \models_T \varphi \\
\Pi, i \models_T \forall\pi.\,\varphi & \text{iff} & \forall t \in T.\, \Pi[\pi \mapsto t], i \models_T \varphi \\
\Pi, i \models_T \exists q.\,\varphi & \text{iff} & \exists t_q \in (2^{\{q\}})^\omega.\, \Pi, i \models_{T[q \mapsto t_q]} \varphi \\
\Pi, i \models_T \forall q.\,\varphi & \text{iff} & \forall t_q \in (2^{\{q\}})^\omega.\, \Pi, i \models_{T[q \mapsto t_q]} \varphi\ .
\end{array}
$$

The semantics of propositional quantification is defined so that in the scope of a quantifier binding q, all traces agree on their q-sequence.

3.2 HyperQPTL Satisfiability

In this section, we determine the undecidability borders for the satisfiability problem of HyperQPTL.

Definition 9 (HyperQPTL satisfiability)**.** *A HyperQPTL formula φ is satisfiable if there exists a non-empty set of traces T such that $\emptyset, 0 \models_T \varphi$, where $\emptyset$ is the empty trace assignment.*

We use the following notation to define HyperQPTL fragments. We write $\forall_\pi$ and $\forall_q$ for a single universal trace and propositional quantifier, respectively. To denote a sequence of universal trace and propositional quantifiers, we write $\forall^*_\pi$ and $\forall^*_q$. Furthermore, we use $\forall^*_{\pi/q}$ for a sequence of mixed universal quantification. We use the analogous notation for existential quantifiers. Lastly, Q^*_π and Q^*_q denote a sequence of mixed universal and existential trace and propositional quantifiers, respectively. As an example, the $\forall^*_\pi Q^*_q$ fragment denotes all formulas of the form $\forall\pi_1.\dots\forall\pi_m.\exists/\forall q_1.\dots\exists/\forall q_n.\,\varphi$, where φ is quantifier free. The results of this section are summarized in Figure 3.2.

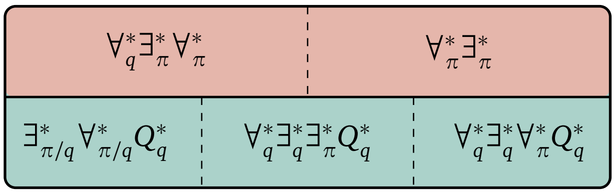

$\forall^*_q \exists^*_\pi \forall^*_\pi$
$\forall^*_\pi \exists^*_\pi$
$\exists^*_{\pi/q} \forall^*_{\pi/q} Q^*_q$
$\forall^*_q \exists^*_q \exists^*_\pi Q^*_q$
$\forall^*_q \exists^*_q \forall^*_\pi Q^*_q$

Figure 3.2: The satisfiability problem of HyperQPTL. Left and below of the solid line are the decidable fragments, right above the solid line the undecidable fragments.

3.2.1 Decidable Fragments

We begin by defining the decidable fragments by their quantifier structure based on the innermost trace quantifier. Propositional quantification to the right of the innermost trace quantifier has no impact on decidability since QPTL is decidable. Note that the decidable fragment does also include the alternation-free fragments where many information-flow properties lie. We first prove that the fragments consisting of a propositional $\forall\exists$ quantifier alternation followed by one type of trace quantifier is decidable. When followed by an existential trace quantifier, we simulate trace quantification with multiple propositional quantification.

Theorem 6. *Satisfiability of the $\forall^*_q \exists^*_q \exists^*_\pi Q^*_q$ fragment of HyperQPTL is decidable.*

Proof. Since QPTL is decidable, it is sufficient to show that HyperQPTL formulas in the $\forall^*_q \exists^*_q \exists^*_\pi$ fragment are decidable. We reduce the HyperQPTL formula to an equisatisfiable QPTL formula as follows. Let a HyperQPTL formula $\forall^*_q \exists^*_q \exists_{\pi_0} \dots \exists_{\pi_{n-1}} \exists_{\pi'}.\ \varphi$ over $a^i \in AP$ be given. We eliminate the existential trace quantification by replacing the trace quantifier $\exists \pi'$ with k existential quantifiers over propositions, one for every atomic proposition in $a^i \in AP$:

$$\forall^*_q \exists^*_q \exists_{\pi_0} \dots \exists_{\pi_{n-1}} \bigexists_{a^i_{\pi'} = a^0_{\pi'}}^{a^{(k-1)}_{\pi'}} a^i_{\pi'}.\varphi \quad .$$

By iteration, the resulting formula is an equisatisfiable QPTL formula, which can be decided. □

When followed by a universal trace quantifier, we can exploit the fact that every trace in the model must satisfy the formula. This means it is sufficient to construct a singleton set. The universal trace quantification thus translates to a preceding existential propositional quantifier simulating an existential trace quantification.

Theorem 7. *Satisfiability of the $\forall^*_q \exists^*_q \forall^*_\pi Q^*_q$ fragment of HyperQPTL is decidable.*

Proof. Since QPTL is decidable, it is sufficient to show that HyperQPTL formulas in the $\forall_q^*\exists_q^*\forall_\pi^*$ fragment are deciable. We reduce the HyperQPTL formula to an equisatisfiable QPTL formula as follows. Let a HyperQPTL formula $\forall_q^*\exists_q^*\forall_\pi^*\varphi$ over $a^i \in AP$ be given. We eliminate all universal trace quantifiers by preceding existential propositional quantifier as follows:

$$\underset{a^i_{\pi'}=a^0_{\pi'}}{\overset{a^{(k-1)}_{\pi'}}{\exists}} a^i_{\pi'}.\forall_q^*\exists_q^*.\ \varphi\ .$$

The resulting formula is an equisatisfiable QPTL formula, which can be decided. □

Furthermore, as long as there is no $\forall\exists$ quantifier alternation of any kind, HyperQPTL formulas can be decided.

Theorem 8. *Satisfiability of the $\exists^*_{\pi/q}\forall^*_{\pi/q}Q^*_q$ fragment of HyperQPTL is decidable.*

Proof. Since QPTL is decidable, it is sufficient to show that HyperQPTL formulas in the $\exists^*_{\pi/q}\forall^*_{\pi/q}$ fragment are decidable. We do so by reducing the problem to the satisfiability problem of QPTL. Let a HyperQPTL formula $\exists^*_{\pi/q}\forall^*_{\pi/q}.\ \varphi$ over $a^i \in AP$ be given. We inductively construct an equirealizable QPTL formula as follows. We consider four cases: 1) The quantifier prefix is empty, the formula is then a QPTL formula. 2) If the innermost quantifier is a propositional quantification, we leave it unchanged. 3) The innermost quantifier is an existential trace quantifier, i.e., let $\exists^*_{\pi/q}\exists_{\pi'}.\ \varphi$ be given. We eliminate the existential trace quantification by replacing the trace quantifier $\exists\pi'$ with k existential quantifiers over propositions, one for every atomic proposition in $a^i \in AP$:

$$\exists^*_{\pi/q} \underset{a^i_{\pi'}=a^0_{\pi'}}{\overset{a^{(k-1)}_{\pi'}}{\exists}} a^i_{\pi'}.\ \varphi\ .$$

4) The innermost quantifier is a universal trace quantifier, i.e., let $\exists^*_{\pi/q}\forall^*_{\pi'/q}.\ \varphi$ with n existential and m universal trace quantifier be given. We eliminate the universal quantification by explicitly enumerating every possible interaction between the universal and existential quantifiers, a technique already used to prove the decidability of the $\exists^*\forall^*$ HyperLTL fragment [70]:

$$\exists^*_{\pi/q}.\bigwedge_{j_1=0}^{n-1}\cdots\bigwedge_{j_m=0}^{n-1}\varphi[\pi_{j_1}/\pi'_1]\ldots\varphi[\pi_{j_m}/\pi'_m]\ ,$$

where $\varphi[\pi_j/\pi']$ denotes that the trace variable π'_i in φ is replaced by π_j. This construction results in a QPTL formula φ_{QPTL}, which is equisatisfiable to φ. □

3.2.2 Undecidable Fragments

In this section, we determine the undecidable fragments of HyperQPTL. We first note that HyperQPTL inherits the undecidable fragment of HyperLTL.

Corollary 2. *Satisfiability of the $\forall^*_\pi\exists^*_\pi$ fragment of HyperQPTL is undecidable.*

We now prove that formulas in the $\forall^*_q\exists^*_\pi\forall^*_\pi$ fragment are undecidable with a reduction from Post's Correspondence Problem (PCP) [169].

Theorem 9. *Satisfiability of the $\forall^*_q\exists^*_\pi\forall^*_\pi$ fragment of HyperQPTL is undecidable.*

Proof. We give a reduction from PCP to a HyperQPTL formula from the $\forall^*_q\exists^*_\pi\forall^*_\pi$ fragment. In PCP, we are given two equally long lists α and β consisting of finite words from some alphabet Σ of size n. PCP is the problem to find an index sequence $(i_k)_{1\leq k\leq K}$ with $K \geq 1$ and $1 \leq i_k \leq n$, such that $\alpha_{i_1}\ldots\alpha_{i_K} = \beta_{i_1}\ldots\beta_{i_K}$. Intuitively, PCP is the problem of choosing an infinite sequence of domino stones (with finitely many different stones), where each stone consists of two words α_i and β_i. Let a PCP instance with $\Sigma = \{a_1, a_2, ..., a_n\}$ and two lists α and β be given. We choose our set of atomic propositions as follows: $\text{AP} := (\Sigma \cup \{\dot{a}_1, \dot{a}_2, ..., \dot{a}_n\} \cup \#)^2$, where we use the dot symbol to encode that a stone starts at this position of the trace. We write $\tilde{a}$ to denote either a or $\dot{a}$. We use $*$ as an arbitrary symbol of the alphabet. The notation $\pi = \vec{q}$ denotes that for every $q_a \in \vec{q}$, it holds that $a_\pi \leftrightarrow q_a$. The premise $(\Box\pi = \vec{q})$ ensures that the propositions $\vec{q}$ are chosen to represent actual traces from the model. We encode the PCP instance into a HyperQPTL formula that is realizable if and only if the PCP instance has a solution:

$$\forall\vec{q}\exists\pi_s\exists\pi'\forall\pi.((\bigvee_{i=1}^{n}(\dot{a}_i,\dot{a}_i)_{\pi_s}) \wedge (\bigvee_{i=1}^{n}(\tilde{a}_i,\tilde{a}_i)_{\pi_s})\,\mathcal{U}\,\Box(\#,\#)_{\pi_s})$$
$$\wedge (\Box\vec{q} = \pi) \rightarrow \varphi_{stone\&shift}(\pi,\pi') \ .$$

With $\forall\vec{q}$, we simulate universal trace quantification by quantifying over all propositions in the alphabet by q_a for each $a \in \Sigma$. The first conjunct encodes that there exists a solution trace π_s to the PCP problem, that represents lists α and β. The second conjunct $\varphi_{stone\&shift}(\pi',\pi)$ is a disjunction that encodes that the trace simulated by $\vec{q}$, i.e., π starts with a valid encoding of a stone from the PCP instance (or is the empty stone sequence) and that the trace π' encodes the same trace but with the first stone removed [70].

For example, let α with $\alpha_1 = a$, $\alpha_2 = ab$, $\alpha_3 = bba$, and β with $\beta_1 = baa$, $\beta_2 = aa$ and $\beta_3 = bb$ be given. A possible solution for this PCP instance is $(3,2,3,1)$, since $bbaabbbaa = i_\alpha = i_\beta$. For example, the third stone is encoded as follows:

$$\begin{aligned}\Big(\Big(((\dot{b},\dot{b})_\pi \wedge \bigcirc(b,b)_\pi \wedge \bigcirc\bigcirc(a,\dot{*})_\pi \wedge \bigcirc\bigcirc\bigcirc(\dot{*},\tilde{*})_\pi)\\ \vee ((\dot{b},\dot{b})_\pi \wedge \bigcirc(b,b)_\pi \wedge \bigcirc\bigcirc(a,\#)_\pi \wedge \bigcirc\bigcirc\bigcirc(\#,\#)_\pi)\Big)\\ \wedge \Box(\bigcirc\bigcirc\bigcirc(\tilde{a},*)_\pi \rightarrow (\tilde{a},*)_{\pi'})\\ \wedge \Box(\bigcirc\bigcirc\bigcirc(\tilde{b},*)_\pi \rightarrow (\tilde{b},*)_{\pi'})\\ \wedge \Box(\bigcirc\bigcirc\bigcirc(\#,q_*)_\pi \rightarrow (\#,*)_{\pi'})\\ \wedge \Box(\bigcirc\bigcirc(*,\tilde{a})_\pi \rightarrow (*,\tilde{a})_{\pi'})\\ \wedge \Box(\bigcirc\bigcirc(*,\tilde{b})_\pi \rightarrow (*,\tilde{b})_{\pi'})\\ \wedge \Box(\bigcirc\bigcirc(*,\#)_\pi \rightarrow (*,\#)_{\pi'})\Big)\ .\end{aligned}$$

The trace π_s represents the solution:

$$(\dot{b},\dot{b})(b,b)(a,\dot{a})(\dot{a},a)(b,\dot{b})(\dot{b},b)(b,\dot{b})(a,a)(\dot{a},a)(\#,\#)(\#,\#)\ldots$$

The $\forall\exists$ quantifier structure requires that there exists another trace, with the first stone removed:

$$(\dot{a},\dot{a})(b,a)(\dot{b},\dot{b})(b,b)(a,\dot{b})(\dot{a},a)(\#,a)(\#,\#)(\#,\#)\ldots$$

Continuing this, results in the following traces:

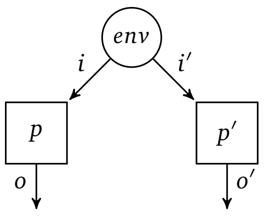
$$\begin{aligned}&(\dot{b},\dot{b})(b,b)(a,\dot{b})(\dot{a},a)(\#,a)(\#,\#)(\#,\#)\ldots\\ &(\dot{a},\dot{b})(\#,a)(\#,a)(\#,\#)(\#,\#)\ldots\\ &(\#,\#)(\#,\#)\ldots\end{aligned}$$

These traces verify the solution provided on the trace π_s by removing one stone after the other. Thus, the formula is satisfiable iff the PCP instance has a solution. □

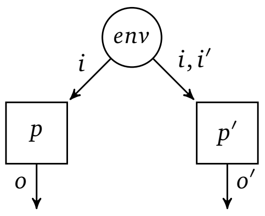
We, thus, identified the decidable fragments of HyperQPTL.

3.3 HyperQPTL Realizability

In this section, we proceed our analysis of HyperQPTL by considering the realizability problem of HyperQPTL. We determine the decidability borders and provide experimental results on a prototype implementation capable of synthesizing arbiters that respect promptness.



(a) Information fork: An architecture with two processes; process p to produces output o from input i and p' produces output o' from input i'.



(b) No information fork: The same architecture as on the left, where the inputs of process p' are changed to i and i'.

Figure 3.3: Distributed architectures

Definition 10 (HyperQPTL Realizability). *A HyperQPTL formula φ over atomic propositions $AP = I \dot{\cup} O$ is realizable if there is a strategy $f : (2^I)^* \to 2^O$ that satisfies φ.*

For technical reasons, we assume (without loss of generality) that quantified atomic propositions are classified as outputs, not inputs. This complies with the intuition that propositional quantifiers should be a means for additional expressiveness; they should not overwrite the inputs received from the environment.

Compared to the standard realizability problem, the distributed realizability problem is defined over an architecture, containing a number of processes interacting with each other. The goal is to find a strategy for each of the processes. In the following proofs, we will make use of the distributed realizability problem of QPTL, which we therefore also define formally. A *distributed architecture* [167, 83] A over atomic propositions AP is a tuple $\langle P, p_{env}, \mathcal{I}, \mathcal{O} \rangle$, where P is a finite set of processes and $p_{env} \in P$ is a designated environment process. The functions $\mathcal{I} : P \to 2^{\mathrm{AP}}$ and $\mathcal{O} : P \to 2^{\mathrm{AP}}$ define the inputs and outputs of processes. The output of one process can be the input of another process. The output of the processes must be pairwise disjoint, i.e., for all $p \neq p' \in P$ it holds that $\mathcal{O}(p) \cap \mathcal{O}(p') = \emptyset$. We assume that the environment process forwards inputs to the processes and has no input of its own, i.e., $\mathcal{I}(p_{env}) = \emptyset$.

Definition 11 (Distributed QPTL Realizability [83]). *A QPTL formula φ over free atomic propositions AP is realizable in an architecture $A = \langle P, p_{env}, \mathcal{I}, \mathcal{O} \rangle$ if for each process $p \in P$, there is a strategy $f_p : (2^{\mathcal{I}(p)})^* \to 2^{\mathcal{O}(p)}$ such that the combination of all f_p satisfies φ.*

The distributed realizability problem for QPTL is (inherited from LTL) in general undecidable [167]. However, we will use the result that the problem remains decidable for architectures without *information forks* [83]. The notion of information forks captures the flow of data in the system. Intuitively, an architecture contains

an information fork if the processes cannot be ordered linearly according to their informedness. Formally, an information fork in an architecture $A = \langle P, p_{env}, \mathcal{I}, \mathcal{O}\rangle$ is defined as a tuple (P', V', p, p'), where p, p' are two different processes, $P' \subseteq P$, and $V' \subseteq AP$ is disjoint from $\mathcal{I}(p) \cup \mathcal{I}(p')$. (P', V', p, p') is an information fork if P' together with the edges that are labeled with at least one variable from V' forms a subgraph rooted in the environment and there exist two nodes $q, q' \in P'$ that have edges to p, p', respectively, such that $\mathcal{O}(q) \cap \mathcal{I}(p) \nsubseteq \mathcal{I}(p')$ and $\mathcal{O}(q') \cap \mathcal{I}(p') \nsubseteq \mathcal{I}(p)$. The definition formalizes the intuition that p and p' receive incomparable input bits, i.e., they have incomparable information. Two example architectures are depicted in Fig. 3.3 [73]. The processes in Fig. 3.3a receive distinct inputs and thus neither process is more informed than the other. The architecture therefore contains an information fork with $P' = \{env, p, p'\}, V' = \{i, i'\}, q = env, q' = env$. The processes in Fig. 3.3b can be ordered linearly according to the subset relation on the inputs and thus the architecture contains no information fork.
In the following sections, we identify tight syntactic fragments of HyperQPTL for which the standard realizability problem is decidable. We give decidability proofs and show that formulas outside the decidable fragments are in general undecidable. An important aspect for decidability is the number of universal trace quantifiers that appear in the formula. We thus present our findings in three categories, depending on the number of universal trace quantifiers a formula has. Figure 3.4 summarizes our results. We establish that a major factor for the decidability of the realizability problem consists in the number of universal trace occurring in a formula. Realizability of HyperQPTL formulas without $\forall\pi$ quantifiers is decidable (Section 3.3.1). Formulas with a single $\forall\pi$ are decidable if they belong to the $\exists^*_{q/\pi}\forall^*_q\forall_\pi Q^*_q$ fragment. This fragment also contains promptness. For more than one universal trace quantifier, we show that decidability can be guaranteed for a fragment that we call the linear $\forall^*_\pi Q^*_q$ fragment. We also show that all the above fragments are tight, i.e., realizability of all other formulas is in general undecidable. Lastly, Section 3.4 presents experiments for the prototype implementation of our bounded synthesis algorithm for HyperQPTL.

3.3.1 No Universal Trace Quantifier

We show that the realizability problem of any HyperQPTL formula without a $\forall_\pi$ quantifier is decidable. The problem is reduced to QPTL realizability.

Theorem 10. *Realizability of the $(\exists^*_\pi Q^*_q)^*$ fragment of HyperQPTL is decidable.*

Proof. Let a $(\exists^*_\pi Q^*_q)^*$ HyperQPTL formula φ over $\text{AP} = I \mathbin{\dot\cup} O = \{a^0, \dots, a^k\}$ with trace quantifiers $\pi_0, \dots \pi_n$ be given. We reduce the problem to the realizability problem of QPTL, which is known to be decidable (since QPTL formulas can be translated to

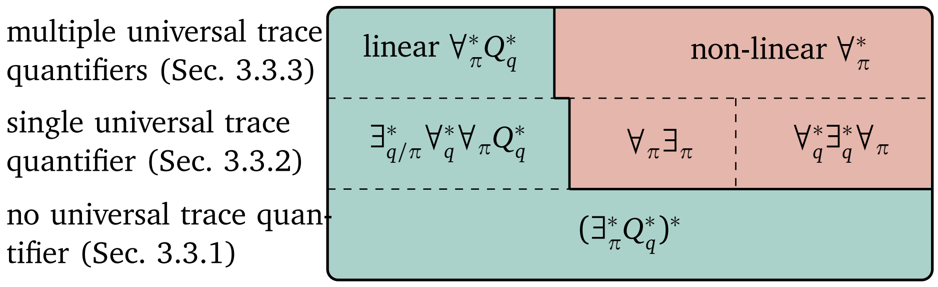

multiple universal trace quantifiers (Sec. 3.3.3)
linear $\forall^*_\pi Q^*_q$
non-linear $\forall^*_\pi$
single universal trace quantifier (Sec. 3.3.2)
$\exists^*_{q/\pi}\forall^*_q\forall_\pi Q^*_q$
$\forall_\pi\exists_\pi$
$\forall^*_q\exists^*_q\forall_\pi$
no universal trace quantifier (Sec. 3.3.1)
$(\exists^*_\pi Q^*_q)^*$

Figure 3.4: The realizability problem of HyperQPTL. Left and below of the solid line are the decidable fragments, right above the solid line the undecidable fragments.

Büchi automata). The idea is to replace each existential trace quantifier $\exists\pi_i$ with quantification of propositions $a^0_{\pi_i}, a^1_{\pi_i}, \dots, a^k_{\pi_i}$, one for each $a^j \in \mathrm{AP}$, thereby mimicking the quantification of a trace. To make sure that only traces from an actual strategy tree are chosen, we add a dependency formula which forces the outputs to be dependent on the inputs. The following QPTL formula implements the idea:

$$\begin{aligned}\varphi_{QPTL} ::= {} & \varphi[i \le n : \exists\pi_i \mapsto \exists a^0_{\pi_i}.\dots\exists a^k_{\pi_i}.] \wedge \\ & \bigwedge_{i\le n}\bigwedge_{j\le n}(I_{\pi_i} \ne I_{\pi_j})\mathcal{R}(O_{\pi_i} = O_{\pi_j}) \ .\end{aligned}$$

We use the notation $[i \le n : \exists\pi_i \mapsto \exists a^0_{\pi_i}.\dots\exists a^k_{\pi_i}.]$ to indicate that each π_i for $0 \le i \le n$ is replaced with the respective series of existential propositional quantification. Furthermore, we write $I_{\pi_i} \ne I_{\pi_j}$ as syntactic sugar for $\bigvee_{a\in I} a_{\pi_i} \not\leftrightarrow a_{\pi_j}$ (and similarly for $O_{\pi_i} = O_{\pi_j}$). We show that φ and φ_{QPTL} are equirealizable. For the first direction, assume that φ is realizable by a strategy f. Notice that all atomic propositions in φ_{QPTL} are bound by a propositional quantifier. Therefore, if the witness sequences for the quantified propositions can be chosen correctly, any strategy realizes φ_{QPTL}. Propositions $a^j_{\pi_i}$ are chosen according to the witness traces of $f \models \varphi$. Witnesses for the remaining atomic propositions are also chosen according to their witnesses from $f \models \varphi$. Now, the first conjunct of φ_{QPTL} is fulfilled since $f \models \varphi$ holds. The second conjunct is fulfilled since any two traces π_i, π_j of a strategy tree fulfill by construction $(I_{\pi_i} \ne I_{\pi_j})\mathcal{R}(O_{\pi_i} = O_{\pi_j})$. For the other direction, assume that φ_{QPTL} is realizable (by construction independently from the strategy). Let $t_{a^0_{\pi_0}}, \dots, t_{a^k_{\pi_n}}$ be the witness sequences for the respective quantified atomic propositions. The following strategy realizes φ:

$$f(\sigma) = \begin{cases} \{t_{a_{\pi_i}}[|\sigma|] \mid a \in O\} & \text{if for some } i \le n, \\ & \quad \sigma = \{t_{a_{\pi_i}}[0] \mid a \in I\}\dots\{t_{a_{\pi_i}}[|\sigma|] \mid a \in I\} \\ \emptyset & \text{otherwise} \ . \end{cases}$$

Strategy f chooses the outputs according to the witnesses for the propositions encoding the traces. Note that because of the second conjunct in φ_{QPTL}, the output is always unique, even if several encoded traces start with the same input sequence. Now, $f \models \varphi$ holds because of the first conjunct of φ_{QPTL}. $\square$

3.3.2 Single Universal Trace Quantifier

In this fragment, we allow exactly one universal trace quantifier. It is particularly interesting as it contains many promptness properties. For example, the following promptness formulation mentioned in the introduction lies within the fragment:

$$\exists b. \forall \pi.\ \diamondsuit b \wedge (\neg b \ \mathcal{U} e_\pi)\ .$$

Theorem 11. *Realizability of the $\exists^*_{q/\pi}\forall^*_q\forall_\pi Q^*_q$ fragment of HyperQPTL is decidable.*

We show the theorem in two steps. First, we generalize a proof from [73], showing that realizability of the $\exists^*_\pi\forall_\pi Q^*_q$ fragment is decidable. Second, we show that we can reduce the realizability problem of any HyperQPTL formula to a formula where some propositional quantifiers are replaced with trace quantifiers.

Lemma 1. *Realizability of the $\exists^*_\pi\forall_\pi Q^*_q$ fragment of HyperQPTL is decidable.*

Proof. The reasoning generalizes the proof in [73] showing that realizability $\exists^*_\pi\forall_\pi$ HyperLTL formulas is decidable. We reduce the problem to the distributed realizability problem of QPTL without information forks, which is, since QPTL is subsumed by the μ-calculus [187, 128], decidable [83]. Let a HyperQPTL formula $\varphi = \exists\pi_1.\dots\exists\pi_n.\forall\pi.\psi$ over $AP = I \dot\cup O$ be given, where ψ is from the Q^*_q fragment. We define a distributed architecture $\mathcal{A}$ over an extended set of atomic propositions $\mathrm{AP}' = I \cup O \cup I' \cup O'$. Similarly to the proof in Theorem 10, I' and O' are composed of a copy of the atomic propositions for each existentially quantified variable π_j. Formally, $I' = \bigcup_{1\le j\le n}\{i_{\pi_j} \mid i \in I\}$ and $O' = \bigcup_{1\le j\le n}\{o_{\pi_j} \mid o \in O\}$. We define $\mathcal{A}$ as follows:

$$\begin{aligned}
\mathcal{A} &::= \langle (p_{env}, p_1, p_2), p_{env}, \mathcal{I}, \mathcal{O}, \rangle \\
\mathcal{I} &::= (p_1 \mapsto \emptyset, p_2 \mapsto I) \\
\mathcal{O} &::= (p_{env} \mapsto I, p_1 \mapsto I' \cup O', p_2 \mapsto O)\ .
\end{aligned}$$

The architecture is displayed in Fig. 3.5. The idea is that process p_1 sets the values of all i_{π_j} and o_{π_j} (for $j \le n$) and thereby determines the choice for the existentially quantified traces. Process p_1 receives no input and therefore needs to make a deterministic choice. Process p_2 then solves the realizability of formula $\forall\pi.\psi$. The following QPTL formula φ' encodes the idea:

$$\varphi' ::= \psi' \wedge (\bigwedge_{1\le j\le n} (I_{\pi_j} \ne I)\mathcal{R}(O_{\pi_j} = O))\ ,$$

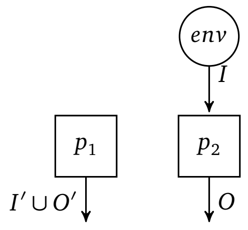


Figure 3.5: Distributed architecture encoding existential choice of traces.

where ψ' is defined as ψ, where all a_π are replaced by a (but atomic propositions a_{π_j} are still part of ψ'!). Note that QPTL formulas implicitly quantify over all traces universally. Similarly to the proof in Theorem 10, the second conjunct ensures that process p_1 encodes actual paths from the strategy tree of process p_2 (which is also the strategy tree for formula φ). Thus, φ' is realizable for the distributed architecture $\mathcal{A}$ iff φ is realizable. □

To state the second lemma, we need to define what it means to replace quantifiers in a formula. Let $\varphi = Q_{\pi/q}, \ldots, Q_{\pi/q}. \psi$ be a HyperQPTL formula, and J be a set of indices such that for all $j \in J$, there exists a propositional quantifier $\exists q_j$ or $\forall q_j$ in φ. Furthermore, assume that no π_j with $j \in J$ occurs in φ and that $a \in$ AP. We denote by $\varphi[J \hookrightarrow_a \pi]$ the formula where each propositional quantifier $\exists q_j$ (or $\forall q_j$, respectively) with $j \in J$ is replaced with the corresponding trace quantifier $\exists \pi_j$ (or $\forall \pi_j$, respectively); and each q_j in ψ is replaced by a_{π_j}.

Lemma 2. *Let any HyperQPTL formula φ over AP $= I \dot\cup O$ and a set of indices J be given. If $\varphi[J \hookrightarrow_i \pi]$ is realizable, then so is φ, where $i \in I$ is an arbitrary input, assuming w.l.o.g., that I is non-empty.*

Proof. Let φ and J be given. Formula $\varphi[J \hookrightarrow_i \pi]$ replaces the quantification over sequences $(2^{\{q\}})^\omega$ with trace quantification, where the trace is only used for statements about a single input i. We thus exploit the fact that in the realizability problem, there is a trace for every input sequence. Therefore, the transformed formula is equirealizable. □

Now, we have everything we need to prove Theorem 11.

Proof of Theorem 11. Let φ be a HyperQPTL formula of the $\exists^*_{q/\pi} \forall^*_q \forall_\pi Q^*_q$ fragment. First, observe that in the quantifier prefix of φ, the $\forall^*_q$ quantifiers and the $\forall_\pi$ can be swapped. The resulting formula belongs to the $\exists^*_{q/\pi} \forall_\pi Q^*_q$ fragment. By Lemma 2, the formula can be transformed to an equirealizable formula of the $\exists^*_\pi \forall_\pi Q^*_q$ fragment, for which realizability is decidable by Lemma 1. □

Lemma 2 allows us to decide realizability of a HyperQPTL formula by replacing propositional quantifiers with trace quantifiers. Thus, we can reduce HyperQPTL realizability to HyperLTL realizability, a fact that we use in Section 3.4 to describe a bounded synthesis algorithm for HyperQPTL.

Corollary 3. *The realizability problem of HyperQPTL can be soundly reduced to the realizability problem of HyperLTL.*

Lastly, we show that the decidable fragment is tight in the class of formulas with a single universal trace quantifier. We do so by showing that a propositional $\forall_q^*\exists_q^*$ quantifier alternation followed by a single trace quantifier $\forall_\pi$ leads to an undecidable realizability problem. The proof is carried out by a reduction from Post's Correspondence Problem.

Theorem 12. *Realizability of any HyperQPTL formula which has a single $\forall_\pi$ quantifier and does not lie in the $\exists_{q/\pi}^*\forall_q^*\forall_\pi Q_q^*$ fragment is undecidable.*

Proof. Inherited from HyperLTL, realizability of formulas with a $\forall_\pi$ quantifier followed by an $\exists_\pi$ quantifier is undecidable [73]. It remains to show that realizability of formulas from the $\forall_q^*\exists_q^*\forall_\pi$ fragment is in general undecidable. We give a reduction from Post's Correspondence Problem (PCP) [169] to a HyperQPTL formula from the $\forall_q^*\exists_q^*\forall_\pi$ fragment. Let a PCP instance with $\Sigma = \{a_1, a_2, ..., a_n\}$ and two lists α and β be given. We choose our set of atomic propositions as follows: AP $::= I \dot{\cup} O$ with $I := \{i\}$ and $O ::= (\Sigma \cup \{\dot{a}_1, \dot{a}_2, ..., \dot{a}_n\} \cup \#)^2$, where we use the dot symbol to encode that a stone starts at this position of the trace. We write $\tilde{a}$ to denote either a or $\dot{a}$. The single input i spans a binary strategy tree. We encode the PCP instance into a HyperQPTL formula that is realizable if and only if the PCP instance has a solution:

$$\forall q_i. \forall \vec{q}. \exists p_i. \exists \vec{p}. \forall \pi. \; ((\Box \pi = p_i) \rightarrow (\Box \pi = \vec{p})) \wedge \\ ((\Box \pi = (q_i, \vec{q})) \rightarrow \varphi_{reduc}(q_i, \vec{q}, p_i, \vec{p})) \;\; ,$$

where $\vec{q}$ and $\vec{p}$ are sequences of universally and existentially quantified propositional variables, such that for each $(o, o') \in O$, there is a $q_{(o,o')} \in \vec{q}$ and a $p_{(o,o')} \in \vec{p}$. Together with q_i and p_i for the input i, they simulate a universally and an existentially quantified trace from the model. The notation $\pi = \vec{q}$ denotes that for every $q_a \in \vec{q}$, it holds that $a_\pi \leftrightarrow q_a$. As seen before, the premise $(\Box \pi = (q_i, \vec{q}))$ and the conjunct $(\Box \pi = p_i) \rightarrow (\Box \pi = \vec{p})$ ensure that the propositions $(q_i, \vec{q})$ and $(p_i, \vec{p})$ are chosen to represent actual traces from the model. The universal quantification π thus only ensures that $(q_i, \vec{q})$ and $(p_i, \vec{p})$, which are used for the main reduction, are chosen correctly. The reduction is implemented in the formula φ_{reduc} and follows the construction in [70], where it is shown that the satisfiability and realizability problem of HyperLTL are undecidable for a $\forall\exists$ trace quantifier prefix:

$$\varphi_{reduc}(q_i, \vec{q}, p_i, \vec{p}) := \varphi_{rel}(q_i) \rightarrow \varphi_{is++}(q_i, p_i) \\ \wedge \varphi_{start}(\varphi_{stone\&shift}(\vec{q}, \vec{p}), q_i) \wedge \varphi_{sol}(q_i, \vec{q}) \;\; ,$$

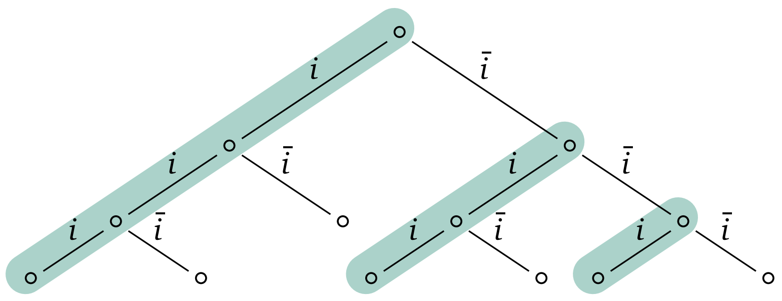


Figure 3.6: A sketch of the strategy tree of our PCP reduction: relevant traces are marked in green.

- $\varphi_{rel}(q_i) := \neg q_i \, \mathrm{U} \Box q_i$ defines the set of *relevant* traces trough the binary strategy tree (see Fig. 3.6).
- $\varphi_{is{+}{+}}(q_i, p_i) := (\neg q_i \wedge \neg p_i) \, \mathrm{U} \, (\Box q_i \wedge \neg p_i \wedge \bigcirc \Box p_i)$ defines that a relevant trace is the direct successor trace of another relevant trace.
- $\varphi_{\mathrm{sol}}(q_i, \vec{q}) ::= \Box q_i \rightarrow ((\bigvee_{i=1}^{n} q_{(\dot{a}_i, \dot{a}_i)}) \wedge (\bigvee_{i=1}^{n} q_{(\tilde{a}_i, \tilde{a}_i)})) \, \mathrm{U} \Box q_{(\#,\#)}$ ensures that the path on which globally i holds is a "solution" trace, i.e., encodes the PCP solution sequence.
- $\varphi_{start}(\varphi, q_i) := \neg q_i \, \mathrm{U}(\varphi \wedge \Box q_i)$ cuts off an irrelevant prefix until φ starts.
- $\varphi_{stone\&shift}(\vec{q}, \vec{p})$ encodes that the trace simulated by $\vec{q}$ starts with a valid encoding of a stone from the PCP instance and that the trace simulated by $\vec{p}$ encodes the same trace but with the first stone removed [70].

The relevant traces verify the solution provided on the $\Box i$ trace by removing one stone after the other. Thus, the formula is realizable iff the PCP instance has a solution. $\Box$

3.3.3 Multiple Universal Trace Quantifiers

When considering multiple universal trace quantifiers $\forall_\pi^*$, the problem becomes undecidable. This is because in HyperLTL, one can encode distributed architectures – for which the problem is undecidable – directly into the formula without using any propositional quantification [73].

Corollary 4. *Realizability of the $\forall_\pi^*$ fragment of HyperQPTL is undecidable.*

However, we show that the realizability problem for formulas with more than one universal trace quantifier is decidable if we restrict ourselves to formulas in the so-called *linear fragment*, i.e., that does not allow an encoding of a distributed architecture. We define the linear fragment of HyperQPTL, where the definitions are adopted from [73].

Let $A, C \subseteq \mathrm{AP}$. We define that atomic propositions $c \in C$ do solely depend on propositions $a \in A$ as the following HyperQPTL formula:

$$D_{A \mapsto C} ::= \forall \pi \forall \pi'. \left(\bigvee_{a \in A} (a_\pi \not\leftrightarrow a_{\pi'}) \right) \mathcal{R} \left(\bigwedge_{c \in C} (c_\pi \leftrightarrow c_{\pi'}) \right) \ .$$

We define a *collapse* function, which collapses a HyperQPTL formula with a $\forall_\pi^*$ universal quantifier prefix into a formula with a single $\forall_\pi$ quantifier. Propositional quantifiers are preserved by the operation. Let φ be $\forall \pi_1 \cdots \forall \pi_n . Q_q^* . \psi$. We define the collapsed formula of φ as $\mathit{collapse}(\varphi) ::= \forall \pi . Q_q^* . \psi[\pi_1 \mapsto \pi][\pi_2 \mapsto \pi] \ldots [\pi_n \mapsto \pi]$ where $\psi[\pi_i \mapsto \pi]$ replaces all occurrences of π_i in ψ with π.

Lemma 3. *Either $\varphi \equiv \mathit{collapse}(\varphi)$ or φ has no equivalent $\forall_\pi^1 . Q_q^*$ formula.*

Proof. The collapse function solely works on the trace quantification mechanism of the HyperQPTL formula, by reducing them to a single universal quantification. The theorem has been proven for $\forall^*$ HyperLTL formulas in [73]. Inner propositional quantification does not interfere with this mechanism, hence, the proof can be carried out identically. $\square$

Now we can formally define the linear $\forall_\pi^*$ fragment. Intuitively, we require that every input-output dependency can be ordered linearly, i.e., we are restricted to linear architectures without information forks (see Figure 3.3).

Definition 12. *Let $O = \{o_1, \ldots, o_n\}$. A HyperQPTL formula φ is called* linear *if for all $o_i \in O$ there is a $J_i \subseteq I$ such that $\varphi \wedge D_{I \mapsto O} \equiv \mathit{collapse}(\varphi) \wedge \bigwedge_{o_i \in O} D_{J_i \mapsto \{o_i\}}$ and $J_i \subseteq J_{i+1}$ for all $i \leq n$.*

This results in the following corollary. Since the universal quantifiers can be collapsed, the resulting problem is the realizability problem of QPTL in a linear architecture, which is decidable [83].

Corollary 5. *Realizability of the linear $\forall_\pi^* Q_q^*$ fragment of HyperQPTL is decidable.*

We identifed the largest possible fragments for which the realizability problem of HyperQPTL remains decidable. The three fragments for which we could prove decidability all subsume the logic QPTL, for which the realizability problem is known to be non-elementary (already its satisfiability problem is non-elementary [197]). Hence, realizability of the discussed HyperQPTL fragments has a non-elementary lower bound.

instance	system bound	$\exists$-strategy bound	result	time [sec.]
arbiter-2-prompt	2	1	unsat	< 1
	2	2	sat	< 1
arbiter-2-full-prompt	3	1	unsat	2.4
	3	2	sat	6.0
arbiter-3-prompt	3	1	unsat	4.2
	3	2	sat	9.5
arbiter-4-prompt	4	1	unsat	97
	4	2	?	TO

Table 3.1: Experimental results for prompt arbiter.

3.4 Experiments

We have implemented a prototype tool that can solve the HyperQPTL realizability problem using the bounded synthesis approach [84]. More concretely, we extended the HyperLTL synthesis tool BoSy [67, 73, 44]. Bosy reduces the HyperLTL synthesis problem to an SMT constraint system which is then solved by z3 [47]. We implemented the reduction of HyperQPTL synthesis to HyperLTL synthesis (Corollary 3) in BoSy, such that the tool can also handle HyperQPTL formulas. We evaluated the tool against a range of benchmarks sets, shown in Table 3.1. The first column indicates the parameterized benchmark name. The second and third columns indicate the bounds given to the bounded synthesis procedure. The second column is the bound on the size of the system. The newest version of BoSy also handles quantifier alternations by viewing them as a two-player game between the $\forall$ player and the $\exists$ player [44]. Existential trace quantification is then replaced by strategic choice via the introduction of prophecy variables. Bounds on the size of the strategy for the existential player are given column three.

We synthesized a range of resource arbiters. Our benchmark set is parametric in the number of clients that can request access to the shared resource (written arbiter-k-prompt where k is the number of clients in Table 3.1). Unlike normal arbiters, we require the arbiter to fulfill promptness for some of the clients, i.e., requests must be answered within a bounded number of steps [202]. We state the promptness requirement in HyperQPTL by applying the *alternating-color technique* from [131]. Intuitively, the alternating-color technique works as follows: We quantify a q-sequence that “changes color" between q and $\neg q$. Each change of color is used as a potential bound. Once a request occurs, the grant must be given withing two changes of color. Thus, the HyperQPTL formulation amounts to the following specifications, here exemplary for 2 clients, where we require promptness only for client 1:

$$\forall\pi. \Box\neg(g^1_\pi \wedge g^2_\pi) \tag{3.4}$$

$$\forall\pi. \Box(r^2_\pi \rightarrow \Diamond g^2_\pi) \tag{3.5}$$

$$\begin{aligned}\exists q. \forall\pi. \Box\Diamond q \wedge \Box\Diamond\neg q \\ \wedge \Box(r^1_\pi \rightarrow (q \rightarrow (q \mathbin{\mathrm{U}} (\neg q \mathbin{\mathrm{U}} g^1_\pi))) \\ \wedge (\neg q \rightarrow (\neg q \mathbin{\mathrm{U}} (q \mathbin{\mathrm{U}} g^1_\pi))))\end{aligned} \tag{3.6}$$

$$\forall\pi. (\neg g^1_\pi \mathbin{\mathrm{W}} r^1_\pi) \wedge (\neg g^2_\pi \mathbin{\mathrm{W}} r^2_\pi) \ . \tag{3.7}$$

Formula 3.4 states mutual exclusion. Formula 3.5 states that client 2 must be served eventually (but not within a bounded number of steps). Formula 3.6 states the promptness requirement for client 1. It quantifies an alternating q-sequence, which serves as a sequence of global bounds that must be respected on all traces π. Then, if client 1 poses a request, the grant must be given within two changes of the value of q. Formula 3.7 is only added in benchmarks named arbiter-k-full-prompt. It specifies that no spurious grants should be given. BoSy was able to successfully synthesize a prompt arbiter.

3.5 Summary

We studied the satisfiability and realizability problem of HyperQPTL. We showed that realizability problem is decidable for HyperQPTL fragments that contain properties like promptness. Propositional quantification does make the satisfiability and realizability problem of hyperlogics harder. More specifically, the HyperQPTL fragment of formulas with a propositional $\forall\exists$ quantifier alternation followed by a single trace quantifier is undecidable in general, even though the projection of the fragment to HyperLTL has a decidable realizability problem. Lastly, we implemented the bounded synthesis problem for HyperQPTL in BoSy. Using BoSy with HyperQPTL specifications, we have been able to synthesize several resource arbiters respecting promptness.

Chapter 4

Logical Methods for Branching-time Temporal Hyperproperties

While linear-time temporal logics like LTL describe properties of individual traces, branching-time temporal logics like CTL and CTL* describe properties of computation trees, where the branches can be inspected by quantifying existentially or universally over paths. There is a long-standing debate of the merits of linear-time and branching-time specifications (e.g. [208, 209]). In summary, linear-time temporal logics are considered more intuitive, whereas branching-time temporal logics have fragments that can be efficiently model checked (e.g. in polynomial time for CTL [38]). We consider the first (and least expressive) temporal logic for branching-time hyperproperties: HyperCTL*, which generalizes CTL* and subsumes HyperLTL.

In contrast to HyperLTL, where quantifiers are only allowed in a *prefix*, HyperCTL* allows for arbitrary use of quantifiers in a formula. This expressive power is typically useful to state that a system can generate secret information [81]. Generating secret information means that there is, at some point, a branching into observably equivalent paths that differ in the values of a secret. For example, this property can be state as the following HyperCTL formula:

$$\exists \pi. \Diamond \exists \pi'. \, (\Box \bigwedge_{a \in P} a_\pi \leftrightarrow a_{\pi'}) \land (\bigcirc \bigvee_{a \in S} a_\pi \not\leftrightarrow a_{\pi'}) \enspace ,$$

where the set of atomic propositions divides into the two disjoint sets of publicly observable propositions P and secret propositions S. Compared to its linear-time counter part, the process equivalence induced by HyperCTL* is bisimulation [81], i.e., no HyperLTL formula can distinguish trees that have the same trace set, but differ in their branching structure. This is, in general, a limitation as nondeterministic choices are reflected in the branching structure.

Satisfiability checking of a branching-time hyperlogic does not only allow for the analysis of specifications, but also for testing example runs. Two traces can be specified directly in a HyperCTL* formula by utilizing the existential path quantification E and the $\bigcirc$ operator. Let τ_0 and τ_1 be the logical representations of two traces. The following HyperCTL* formula is testing these traces for observational determinism:

$$\tau_0 \wedge \tau_1 \wedge \forall \pi'.\forall \pi''.\square(O_{\pi'} \leftrightarrow O_{\pi''}) \ ,$$

where O denotes a set of observable outputs.

We show that the satisfiability problem of HyperCTL* remains decidable for the $\exists^*$ fragment. We show that the models in $\exists^*$ fragment have, in general, a multi dimensional comb-shaped structure. For example, the formula $\exists\pi.\square\exists\pi.\varphi$, requires that there is a tree with one witness trace for π on which a trace π' branches off at every point in time (see Figure 4.1). We show that every model for a formula in the $\exists^*$ fragment has a finite representation by cutting off unnecessary parts in the comb-shaped tree. Based on this result, an indirect scope free $\exists^*\forall^*$ fragment can be defined [105], which is decidable as well. Intuitively, the indirect scope free fragment disallows that a universally quantified trace variable indirectly quantifies over traces that branch of in a later point in time. For example, formulas of the shape $\exists\pi.\square(\exists\pi'.\forall\pi''.\varphi)$ are not indirect scope free. Deciding the indirect scope free $\exists^*\forall^*$ fragment, enables above mentioned testing of example runs. The results in this chapter form the basis of the first satisfiability solver for HyperCTL* [105].

We also leave a short remark on the synthesis problem of HyperCTL*. It is possible to define a linear fragment in linear-time temporal hyperlogics (see [74] and Chapter 3), for which the synthesis problem remains decidable, We argue that HyperCTL* inherits this fragment, but there seems to be no further extension of a linear fragment to the realm of branching-time hyperproperties.

Results in this chapter are based on the satisfiability part in “The Hierarchy of Hyperlogics” [42], which was joined work with Norine Coenen, Bernd Finkbeiner, and Jana Hofmann. The chapter is structured as follows. We first define HyperCTL* formally in Section 4.1 and give more intuition on branching-time properties. We then proceed by considering the satisfiability HyperCTL* in Section 4.2 and provide a short remark on realizability in Section 4.3.

4.1 HyperCTL*

CTL* utilizes path quantifiers E meaning “there exists a path”, and, dually, A meaning “for all paths”. For example, the following formula expresses that there exists a path on which eventually a proposition p must hold:

$$E\diamondsuit p \ .$$

This is not expressible in LTL, which can be seen as a subset of CTL* that is inherently universally quantified, i.e., the LTL formula φ is equivalent to the CTL* formula $A\varphi$.

CTL* Syntax. The syntax of CTL*, where φ denotes state formulas and ψ denotes path formulas, is given as follows:

$$\begin{aligned}\varphi &::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathrm{E}\psi\\ \psi &::= \varphi \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc\psi \mid \psi\,\mathrm{U}\,\psi\ ,\end{aligned}$$

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

The semantics of branching-time logics are defined over trees. A *tree* $\mathcal{T}$ is defined as a partially-ordered infinite set of nodes S, where all nodes share a common minimal element $r \in S$, called the *root* of the tree. Moreover, for every node $s \in S$, the set of its *ancestors* $\{s' | s' < s\}$ is totally-ordered. We say that s' is the *direct ancestor* of s, if $s' < s$, and there is no s'' such that $s' < s'' < s$. A Σ*-labeled tree* is defined as a tree $\mathcal{T}$ equipped with a function $L : S \to \Sigma$, that labels every node with an element from a finite set Σ. For the case that $\Sigma = 2^{AP}$, we say that the tree is *AP-labeled*. A *path* through a tree $\mathcal{T}$ is a sequence $\sigma = s_0, s_1, \ldots$ of direct ancestors in $\mathcal{T}$, i.e., for all s_i, s_{i+1}, node s_i is the direct ancestor of s_{i+1}. A path is called *initial* if s_0 is the root node, which we omit if it is clear from the context. We use the same path manipulation operations as for traces. The set of *paths originating in node* $s \in S$ is denoted by $Paths(\mathcal{T}, s)$. If s is the root node, we simply write $Paths(\mathcal{T})$.

CTL* Semantics. The semantics of CTL* is defined over an *AP*-labeled tree $\mathcal{T}$ with nodes S and labeling function L. Given a node $s \in S$ and a path p in $\mathcal{T}$, we define the semantics of CTL* state and path formulas as follows:

$$\begin{array}{ll}
s \models_{\mathcal{T}} a & \text{iff } a \in L(s)\\
s \models_{\mathcal{T}} \neg\varphi & \text{iff } s \not\models_{\mathcal{T}} \varphi\\
s \models_{\mathcal{T}} \varphi_1 \vee \varphi_2 & \text{iff } s \models_{\mathcal{T}} \varphi_1 \text{ or } s \models_{\mathcal{T}} \varphi_2\\
s \models_{\mathcal{T}} \mathrm{E}\psi & \text{iff } \exists p \in Paths(\mathcal{T}, s).\ p \models_{\mathcal{T}} \psi\\
p \models_{\mathcal{T}} \varphi & \text{iff } p[0] \models_{\mathcal{T}} \varphi\\
p \models_{\mathcal{T}} \neg\psi & \text{iff } p \not\models_{\mathcal{T}} \psi\\
p \models_{\mathcal{T}} \psi_1 \vee \psi_2 & \text{iff } p \models_{\mathcal{T}} \psi_1 \text{ or } p \models_{\mathcal{T}} \psi_2\\
p \models_{\mathcal{T}} \bigcirc\psi & \text{iff } p[1,\infty] \models_{\mathcal{T}} \psi\\
p \models_{\mathcal{T}} \psi_1\,\mathrm{U}\,\psi_2 & \text{iff } \exists i \geq 0.\ p[i,\infty] \models_{\mathcal{T}} \psi_2\\
& \quad \wedge \forall 0 \leq j < i.\ p[j,\infty] \models_{\mathcal{T}} \psi_1\ .
\end{array}$$

For a tree $\mathcal{T}$ and a CTL* formula φ, we write $\mathcal{T} \models \varphi$ if $\mathcal{T}$ has root r, such that $r \models_{\mathcal{T}} \varphi$. Operators can be derived in the same fashion as for LTL. The universal path quantifier $A\varphi$ can be derived as follows: $\neg E \neg \varphi$.

Definition 13 (CTL* Satisfiability)**.** *A CTL* formula φ is satisfiable if there exists a tree $\mathcal{T}$ that satisfies φ.*

Theorem 13 ([210]). *The satisfiability problem of CTL* is 2-EXPTIME-complete.*

HyperCTL* Syntax. Quantification in HyperCTL* ranges over the paths in a tree. Let $\pi \in \mathcal{V}$ be a *path variable* from an infinite supply of path variables $\mathcal{V}$ and let $\exists \pi.\ \varphi$ be the explicit existential *path quantification*. HyperCTL* formulas are generated by the following grammar:

$$\varphi ::= a_\pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathrm{U}\, \varphi \mid \exists \pi.\ \varphi \enspace .$$

HyperCTL* Semantics. The semantics of a HyperCTL* formula is defined with respect to a tree $\mathcal{T}$ and a *path assignment* $\Pi : \mathcal{V} \to \mathit{Paths}(\mathcal{T})$, which is a partial mapping from path variables to actual paths in the tree. The satisfaction relation $\models_\mathcal{T}$ is given as follows:

$$\begin{array}{ll}
\Pi, i \models_\mathcal{T} a_\pi & \text{iff } a \in L(\Pi(\pi)[i]) \\
\Pi, i \models_\mathcal{T} \neg\varphi & \text{iff } \Pi, i \not\models_\mathcal{T} \varphi \\
\Pi, i \models_\mathcal{T} \varphi_1 \vee \varphi_2 & \text{iff } \Pi, i \models_\mathcal{T} \varphi_1 \text{ or } \Pi, i \models_\mathcal{T} \varphi_2 \\
\Pi, i \models_\mathcal{T} \bigcirc\varphi & \text{iff } \Pi, i+1 \models_\mathcal{T} \varphi \\
\Pi, i \models_\mathcal{T} \varphi_1 \,\mathrm{U}\, \varphi_2 & \text{iff } \exists j \geq i.\ \Pi, j \models_\mathcal{T} \varphi_2 \\
 & \qquad \wedge \forall i \leq k < j.\ \Pi, k \models_\mathcal{T} \varphi_1 \\
\Pi, i \models_\mathcal{T} \exists \pi.\varphi & \text{iff } \exists p \in \mathit{Paths}(\mathcal{T}).\ p[0,i] = \varepsilon[0,i] \\
 & \qquad \wedge \Pi[\pi \mapsto p, \varepsilon \mapsto p], i \models_\mathcal{T} \varphi \enspace ,
\end{array}$$

where we use ε to denote the last path that was added to the path assignment Π.

We assume formulas to be given in negated normal form. A HyperCTL* formula in NNF is in the $\exists^*$ and $\forall^*$ fragment, respectively, if it contains exclusively universal or exclusively existential path quantifiers. The union of the two fragments is the alternation-free fragment. The formula is in the $\exists^*\forall^*$ fragment, if there is no existential path quantifier in the scope of a universal path quantifier. It is in the $\forall\exists$ fragment if there is exactly one existential path quantifier in the scope of a single universal path quantifier.

4.2 HyperCTL* Satisfiability

In this section, we study the satisfiability problem of HyperCTL*. We prove that the quantifier alternation-free fragments are decidable.

Definition 14 (HyperCTL* satisfiability). *A HyperCTL* formula φ is satisfiable if there exists a tree $\mathcal{T}$ such that $\emptyset, 0 \models_\mathcal{T} \varphi$, where $\emptyset$ is the empty trace assignment.*

4.2.1 The Universal Fragment of HyperCTL*

Lemma 1. *Satisfiability of the $\forall^*$ fragment of HyperCTL* is decidable.*

Proof. We reduce the satisfiability problem of $\forall^*$ HyperCTL* to the satisfiability problem for CTL* of models that are linear trees, i.e., trees having only a single path. $\square$

4.2.2 The Existential Fragment of HyperCTL*

We now prove in two steps that the $\exists^*$ fragment is decidable, regardless of the temporal modalities and the nesting depth of the existential quantifiers. We start with formulas of a specific form and then generalize the result to the full fragment. Subsequently, we establish that the satisfiability problem for HyperCTL* formulas in the full $\exists^*\forall^*$ fragment remains decidable.

Lemma 2. *The satisfiability problem for HyperCTL* formulas of the form*

$$\varphi ::= \exists\pi.(\exists\pi'.\psi')\mathcal{R}(\exists\pi''.\psi'') \ ,$$

where ψ' and ψ'' are quantifier free, is decidable.

Proof Idea. The key idea of the proof is to show that every model of a φ-shaped formula has a finite representation. More concretely, we show that we can represent an arbitrary model $\mathcal{T}$ satisfying φ as a tree $\mathcal{T}_{\mathit{fin}}$ of bounded size. We then show that $\mathcal{T}_{\mathit{fin}}$ can be extended to an infinite tree $\tilde{\mathcal{T}}$ which satisfies φ. We conclude by describing a naive decision procedure which enumerates all bounded trees $\mathcal{T}_{\mathit{fin}}$ and checks whether they can be extended into an infinite model $\tilde{\mathcal{T}}$ for φ. We first give some intuition on how to construct $\mathcal{T}_{\mathit{fin}}$ out of $\mathcal{T}$. Assume a formula $\exists\pi.\square(\exists\pi''.\psi)$, which belongs to the fragment described in Lemma 2 and a model $\mathcal{T}$ satisfying it. In this model, there needs to be a path p witnessing π, and at each point in time i, there must be a path p_i, which branches off of p and serves as a witness for π'' at point in time i. Extracting these witnesses from the model results in a comb-like structure as depicted in Figure 4.1. Through formula ψ, each pair of nodes $p[i+j]$ and $p_i[j]$ are related with each other, e.g., if $\psi = \square(a_\pi \leftrightarrow a_{\pi'})$, then all $p[i+j]$ and $p_i[j]$ must agree on a. The nodes $p[i+j]$ and $p_i[j]$ always reside on the same diagonal in the comb. Like this, all nodes on the same diagonal are related with each other through p. When transforming the witnesses, it is therefore important to only consider one diagonal as a whole and to not alter just a single node on it. Diagonal D_3 is also depicted in Figure 4.1. Next, note that ψ can be transformed into a Büchi automaton which accepts each pair of witness paths $(p[i,\infty],p_i)$. We label each p_i in the comb with the corresponding accepting automaton run. Now, the crucial observation is that if two diagonals in the comb are labeled with the same set of automaton states, we can then *cut out* the part between those two diagonals and

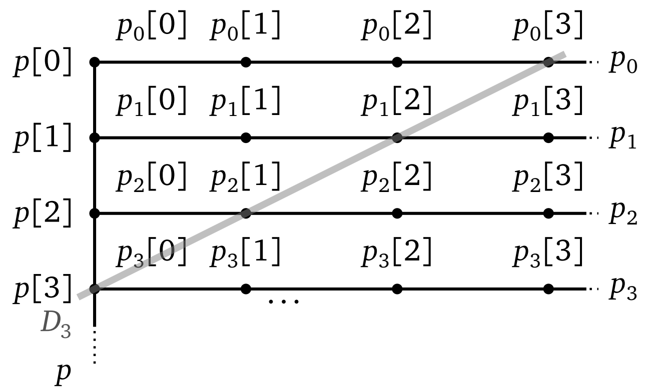

$p_0[0]$ $p_0[1]$ $p_0[2]$ $p_0[3]$
$p_1[0]$ $p_1[1]$ $p_1[2]$ $p_1[3]$
$p_2[0]$ $p_2[1]$ $p_2[2]$ $p_2[3]$
$p_3[0]$ $p_3[1]$ $p_3[2]$ $p_3[3]$
$p[0]$
$p[1]$
$p[2]$
$p[3]$
D_3
p
p_0
p_1
p_2
p_3
...

Figure 4.1: ©IEEE 2019. The witness p for π and the sequence of witnesses p_i for π'' arranged in a comb-like structure, where nodes $p_3[0], p_2[1], p_1[2]$, and $p_0[3]$ reside on the diagonal D_3.

still have accepting runs, i.e., the resulting paths p and all p_i are still witnesses for π and π''. The proof proceeds by repeatedly cutting out nonessential parts of the comb until it has a suitable prefix of bounded size which we call $\mathcal{T}_{fin}$. □

Proof. We assume w.l.o.g. that no $\bigcirc$-modalities occur in the formula; any $\bigcirc$-modality can only add an offset to the operations which we describe in the following. Assume a tree $\mathcal{T}$ over nodes S with labeling function L that satisfies φ, i.e., there exist a path p through $\mathcal{T}$ serving as witness for π. By the semantics of the $\mathcal{R}$-modality, either (Case 1) an infinite sequence $p_0, p_1, p_2, \ldots$ of witnesses for π'' or (Case 2) a finite sequence $p_0, p_1, \ldots, p_n$ of witnesses for π'' and a final witness p' for π'.

Case 1. We construct a Büchi automaton $A_{\psi''}$ that accepts all possible pairs of paths (p, p_i) that satisfy ψ''. The formula ψ'' is quantifier-free and is thus interpreted as an LTL formula where each atomic proposition a_π is as a unique LTL proposition. Let $A'_{\psi''} = (Q', q'_0, \Sigma', \delta', F')$ be the nondeterministic Büchi automaton obtained from ψ'' [212]. We transform $A'_{\psi''}$ into a nondeterministic Büchi automaton $A_{\psi''} = (Q, q_0, \Sigma, \delta, F)$, which reasons separately over π and π'':

- $Q : Q'$,
- $q_0 : q'_0$,
- $\Sigma : S \times S$,
- $\delta : Q \times \Sigma \to 2^Q$ where $q' \in \delta(q, (s_0, s_1))$ iff $q' \in \delta'(q, A)$ and $L(s_0) = \{a \in AP \mid a_\pi \in A\}$ and $L(s_1) = \{a \in AP \mid a_{\pi''} \in A\}$, where $A \subseteq \{a_\pi, a_{\pi''} \mid a \in AP\}$, and
- $F : F'$.

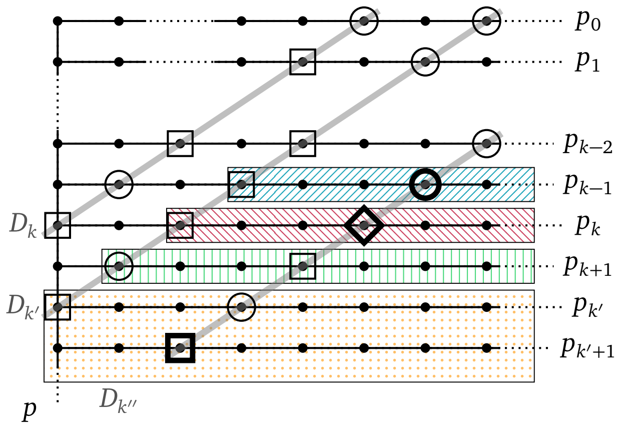


Figure 4.2: ©IEEE 2019. A comb structure with highlighted diagonals D_k, $D_{k'}$, and $D_{k''}$, together with their associated automaton states (depicted as square, circle and diamond). States at representative positions are printed in bold, suffices used for the cut are highlighted.

Note that $A_{\psi''}$ reasons over pairs of paths, while $A'_{\psi''}$ reasons over traces. The automaton $A_{\psi''}$ yields accepting runs r_i for all pairs of witnesses $(p[i,\infty], p_i)$. We can thus *associate* with each node $p_i[j]$ the automaton state $r_i[j]$.

Frontiers. We arrange the witness paths p and $p_0, p_1, p_2, \ldots$ in a comb-like structure as shown in Figure 4.1. For all $k \in \mathbb{N}$, we address the sequence of nodes $p_0[k-0], \ldots, p_k[k-k]$ as the k-th diagonal of the comb, denoted by D_k. We use the usual sequence notation for diagonals, e.g., $D_k[i]$ to address the i-th element of the sequence. We call the set of automaton states associated with nodes in D_k *frontier* F_k, formally $F_k ::= \{r_i[k-i] \mid i \leq k\}$. Note that for every k, $F_k \subseteq Q$.

Cuts. For two diagonals D_k and $D_{k'}$ with $F_k = F_{k'}$, a cut modifies the comb in such a way that the suffix of every node in D_k is replaced by the suffix of a node in $D_{k'}$, where both nodes have to be associated with the same automaton state. Formally, we replace every $p_i[k-i,\infty]$ with some $p_{i'}[k'-i',\infty]$, requiring that $D_k[i]$ and $D_{k'}[i']$ are associated with the same state. Additionally, to preserve the relation of the modified paths with p, we replace the sub-comb with origin in $p[k]$ with the sub-comb with origin in $p[k']$. Note that because of the requirement that $D_k[i]$ and $D_{k'}[i']$ are associated with the same state, the modified witness paths still have accepting runs through $A_{\psi''}$. For $k \leq k'$, we say that two diagonals D_k, $D_{k'}$ with $F_k = F_{k'}$ are frontier-preserving cuttable (for short: cuttable) if for every $q \in F_{k'}$, q is either associated with at least as many nodes on D_k as on $D_{k'}$, or it is associated with $|Q|$ nodes on D_k. For $k \leq k' \leq k''$, a cut preserving $F_{k''}$ is a cut between two cuttable diagonals D_k and $D_{k'}$, such that the set $F_{k''}$ is not modified by the operation. For each $q \in F_{k''}$, pick a position $i_q \leq k''$ as a representative such that the state associated with

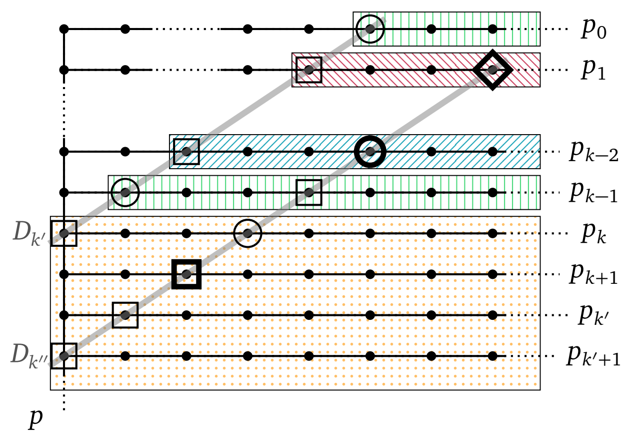


Figure 4.3: ©IEEE 2019. The result of the cutting operation prepared in Figure 4.2. The highlights show which suffix was shifted to which node in the comb.

$D_{k''}[i_q]$ is q. All states q with representative position $i_q \geq k'$ will not be affected by the cut. For representative positions $i_q < k'$, ensure that when choosing suffices from $D_{k'}$ for the cut, each suffix $p_{i_q}[k'-i_q, \infty]$ is chosen at least once. This is possible since we require D_k and $D_{k'}$ to be cuttable. Like this, we ensured that all representative states are not deleted by the cut. Figure 4.2 and Figure 4.3 show the choice of representative positions in a comb and the resulting preserving cut.

Construct $\mathcal{T}_{fin}$. We describe how to perform a series of preserving cuts to ensure that sufficiently many accepting states can be found in a bounded-size prefix $\mathcal{T}_{fin}$ of the comb. First, note that there are at most $2^{|Q|}$ different frontiers. Furthermore, there are at most $c = (|Q|+1)^{|Q|}$ many different equivalence classes of the cuttable property, i.e. for $c+1$ many diagonals, at least two are cuttable. We say that a diagonal D_k is *close* to $D_{k'}$ if $|k'-k| \leq c$. By the pigeonhole principle, for every two diagonals $D_k, D_{k'}$ and state set $F_{k''}$ with $k \leq k' \leq k''$, we can perform a number of cuts on diagonals situated between D_k and $D_{k'}$, each preserving $F_{k''}$, such that at the end, $D_{k'}$ is close to D_k and the set $F_{k''}$ did not change.

There are only finitely many different frontiers in the infinite comb, so at least one frontier occurs on infinitely many diagonals. We call that frontier F_ω. Pick the smallest number $inf \in \mathbb{N}$ such that $F_{inf} = F_\omega$ and cut diagonal D_{inf} as close as possible to D_0 while preserving F_{inf}. Note that the first $|Q|^{|Q|}$ diagonals are, in general, not cuttable; therefore, in the worst case, D_{inf} will be cut close to $D_{|Q|^{|Q|}}$. As a result of these cuts, frontier F_ω might not occur infinitely often anymore. More concretely, diagonals which were previously associated with frontier F_ω will now have frontiers which are a subset of F_ω. Since there are only finitely many different subsets of any finite set, we know that there exists at least one frontier $F_{\omega'} \subseteq F_\omega$ that occurs on infinitely many diagonals.

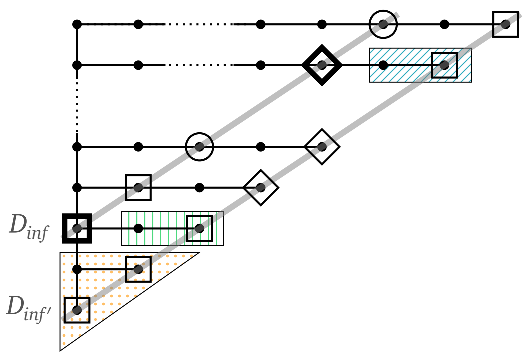


Figure 4.4: ©IEEE 2019. The finite prefix $\mathcal{T}_{fin}$ with diagonals D_{inf} and $D_{inf'}$. States in designated positions on D_{inf} are printed in bold. Suffices that will be copied to extend the prefix comb are highlighted.

For every automaton state $q \in F_{\omega'}$, there exists by construction an $i_q \leq inf$ such that $D_{inf}[i_q]$ is associated with q. We call the set of all i_q the set of designated positions P. Now, find the smallest $inf' > inf$, such that $F_{inf'} = F_{\omega'}$, and for all $i \in P$, r_i has an accepting state between inf and inf'. Such an inf' exist because of the Büchi acceptance condition. We now perform a series of cuts to cut $D_{inf'}$ as close to D_{inf} as possible, each of which preserves $F_{inf'}$. Find the $i \in P$ whose accepting state is closest to D_{inf} and cut the corresponding diagonal close to D_{inf}. Continue with the $i' \in P$ whose accepting state comes next and cut it close to the last diagonal that was cut close. Proceed, until the diagonal of the last accepting state of designated position was cut close. Finally, cut $D_{inf'}$ close to that last diagonal.

We choose $\mathcal{T}_{fin}$ to be the finite prefix of the resulting comb up to (and including) $D_{inf'}$. The depth of $\mathcal{T}_{fin}$ is bounded by $b = |Q|^{|Q|} + (2+|Q|)\cdot(|Q|+1)^{|Q|}$. This is because $\mathcal{T}_{fin}$ consists of a prefix of diagonals up to the first cuttable diagonal (at the most $|Q|^{|Q|}$ many), followed by D_{inf}. Then, $|P| \leq |Q|$ many diagonals have been cut close and the distance between them is at the most $(|Q|+1)^{|Q|}$. Lastly, $D_{inf'}$ was cut close, again with a maximal distance of $(|Q|+1)^{|Q|}$.

Decision Algorithm. We now extend $\mathcal{T}_{fin}$ into an infinite tree $\tilde{\mathcal{T}}$ also satisfying φ. By construction, for each $i \in P$, run r_i has an accepting state between D_{inf} and $D_{inf'}$. Furthermore, for each $q \in F_{inf'}$, there is a designated position $i_q \in P$ such that $D_{inf}[i_q]$ is associated with q. We extend $\mathcal{T}_{fin}$ by extending each node in $D_{inf'}$ as follows: For each $i \leq inf'$ with q associated to $D_{inf'}[i]$ and designated position i_q, we append a copy of $p_{i_q}[inf - i_q + 1, inf' - i_q]$ to $p_i[inf' - i]$. Additionally, we copy the sub-comb starting in node $p[inf+1]$ and append it to node $p[inf']$, thus completing the extension. By construction, we now have a larger finite comb ending in a diagonal $D_{inf''}$ with $F_{inf''} \subseteq F_{inf'}$. Figure 4.4 shows a possible prefix comb $\mathcal{T}_{fin}$ and Figure 4.5 shows how it is extended. Repeating this process indefinitely, we get an infinite,

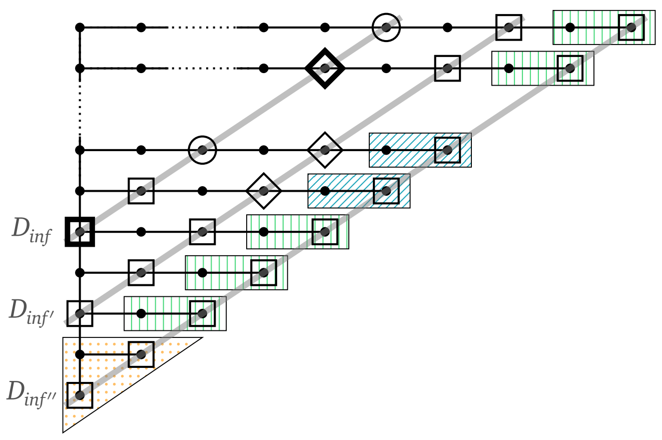


Figure 4.5: ©IEEE 2019. The resulting larger finite prefix after extending every witness path in Figure 4.4 once. The highlights show which part of $\mathcal{T}_{fin}$ was used to extend the prefix.

ultimately periodic model $\tilde{\mathcal{T}}$ where each pair $(p[i,\infty], p_i)$ of witness paths in the comb of $\tilde{\mathcal{T}}$ is accepted by $A_{\psi''}$. It is thus a model for φ.

Case 2. In the case where the release modality is witnessed by path p for π and a sequence of paths $\{p_0, p_1, \dots, p_n, p'\}$ for π'' and π', we proceed very similar to Case 1. We again arrange the witnesses in a comb-like graph, with the only difference that at $p[n]$, there are the two witnesses p_n and p' branching from p. In order to get the same structure as in Case 1, we zip p' and p_n into one witness path $\bar{p}_n$. Furthermore, for all $m > n$, we add dummy witnesses $p_\top = \emptyset^\omega$ branching from p at $p[m]$.

Automata construction. As in Case 1, we associate the paths with the corresponding automaton runs. For $(p[i,\infty], p_i)$ with $i < n$, we use the automaton $A_{\psi''}$, as in Case 1. For $(p[i,\infty], p_\top)$ with $i > n$, we use the automaton $A_\top$, which unconditionally accepts every pair of traces. For $(p[n,\infty], \bar{p}_n)$, we construct a new automaton $A_{\psi'\wedge\psi''}$ based on the LTL automaton $A'_{\psi'\wedge\psi''}$ for $\psi' \wedge \psi''$, similar to the construction of $A_{\psi''}$. For the automaton $A_{\psi'\wedge\psi''}$, we have the following:

$$
\begin{aligned}
&\Sigma : S \times (S \times S) \textit{ and}\\
&\delta : Q \times \Sigma \to 2^Q\textit{, where } q' \in \delta(q, (s_0, (s_1, s_2))) \textit{ iff } q' \in \delta'(q, A),\\
&\textit{ and } L(s_0) = \{a \in AP \mid a_\pi \in A\}, L(s_1) = \{a \in AP \mid a_{\pi'} \in A\},\\
&\textit{ and } L(s_2) = \{a \in AP \mid a_{\pi''} \in A\}\ .
\end{aligned}
$$

We denote the set of states of automaton $A_{\psi''}$ with Q and the set of states of automaton $A_{\psi'\wedge\psi''}$ with $Q_{\psi'\wedge\psi''}$.

Construct $\mathcal{T}_{fin}$. First, cut diagonal D_n close to diagonal D_0. Following D_n, there are again finitely many different cuttable diagonals (containing states from all three automata). Proceeding as in Case 1, construct $\mathcal{T}_{fin}$ such that after D_n, there are two diagonals D_{inf} and $D_{inf'}$ with sufficiently many accepting states in between. The bound b' on the depth of $\mathcal{T}_{fin}$ is obtained analogously to the bound in Case 1. We only remark that n is bounded by $|Q|^{|Q|} + (|Q|+1)^{|Q|}$, and the maximal number of different cuttable diagonals is described in terms of the number of states of all three automata, i.e., $(|Q| + |Q_{\psi' \wedge \psi''}| + |Q_{\top}| + 1)^{(|Q|+|Q_{\psi' \wedge \psi''}|+|Q_{\top}|)}$. We conclude by noting that b' can be used as an over-approximation of bound b in Case 1. Finally, as in Case 1, we construct an infinite satisfying tree $\tilde{\mathcal{T}}$ using $\mathcal{T}_{fin}$.

Decision Algorithm. Enumerate all comb-like prefixes $\mathcal{T}_{fin}$ of bounded depth b' to find a suitable prefix (for either of both cases). Whether a prefix is suitable or not can be decided by labeling it with corresponding runs from the automata of Cases 1 and 2 and checking whether it contains a segment between two diagonals D_{inf} and $D_{inf'}$ which qualifies to be extended into a model $\tilde{\mathcal{T}}$ as described above. When associating the comb prefix with runs from the automata, we have to take into account all finitely-many points in time n where ψ'' could be released by ψ'. If some prefix $\mathcal{T}_{fin}$ is suitable, φ is satisfiable (namely by the described tree $\tilde{\mathcal{T}}$). As shown above, there is a suitable finite prefix of bounded depth b' whenever φ is satisfiable. □

Corollary 6. *The satisfiability problem for HyperCTL* formulas of the form*

$$\exists \pi.(\exists \pi''.\psi'')\,\mathrm{U}(\exists \pi'.\psi') \ ,$$

where ψ' and ψ'' are quantifier free, is decidable.

Proof. We proceed similarly to Case 2 in the proof above. The only difference is that formula ψ'' does not have to hold at the same point in time n where formula ψ' holds. Therefore, the resulting comb does not have two witnesses branching from $p[n]$ that we have to zip. We use an automaton $A_{\psi'}$ instead of $A_{\psi' \wedge \psi''}$ to obtain the run r_n for $(p[n,\infty], p')$. □

We lift the arguments of the above proof to arbitrary formulas in the existential fragment of HyperCTL*.

Lemma 3. *Satisfiability of the $\exists^*$ fragment of HyperCTL* is decidable.*

Proof. Define the existential quantifier depth of a $\exists^*$HyperCTL* formula as the maximal number of alternations between existential quantifiers and the temporal modalities $\mathcal{R}$ and U in the syntax tree. The witnesses of a formula with quantifier depth d can be arranged as a d-dimensional comb. We assume, again, w.l.o.g. that no $\bigcirc$-modality occurs in the formula. Lemma 2 and Corollary 6 cover the case where the comb is 2-dimensional. We now lift the arguments to the general case. Given a d-dimensional comb, we associate the innermost witnesses with the corresponding runs on the d-tuple automata, which we build as before from the inner LTL formulas. A 3-dimensional comb and the corresponding automaton runs are exemplarily depicted in Figure 4.6.

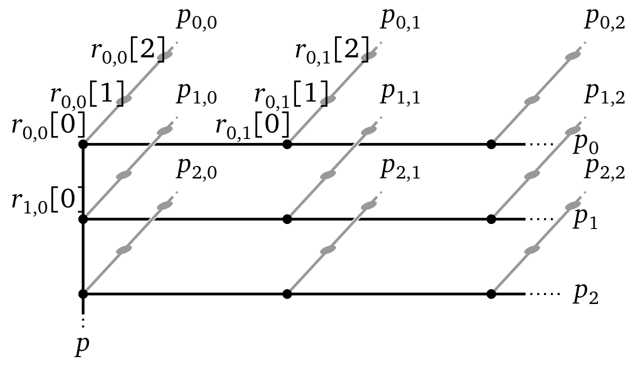


Figure 4.6: ©IEEE 2019. A 3-dimensional comb graph resulting from the arrangement of witnesses for a HyperCTL* formula $\exists\pi.\square(\exists\pi'.\square(\exists\pi''.\varphi))$. The innermost witnesses $p_{i,j}$ are labeled with the automaton states of the corresponding automaton runs.

In the d-dimensional case, diagonals are hyperplanes. We represent the k-th plane D_k in the d-dimensional comb by a nested sequence of depth $d-1$:

$$\begin{aligned}
D_k &::=[D_{k,0},\ldots,D_{k,k}]\\
D_{k,i_1} &::=[D_{k,i_1,0}\ldots,D_{k,i_1,k-i_1}]\\
&\vdots\\
D_{k,i_1,\ldots,i_{d-2}} &::=[p_{i_1,\ldots,i_{d-2},0}[s],p_{i_1,\ldots,i_{d-2},1}[s-1],\ldots\\
&\qquad p_{i_1,\ldots,i_{d-2},s}[0]],\text{ where } s=k-(i_1+\ldots+i_{d-2}) \quad .
\end{aligned}$$

We additionally define d-dimensional frontiers as nested sets of automata states:

$$\begin{aligned}
F_k &::=\{F_{k,i_1}\mid i_1\leq k, i_1\in\mathbb{N}\}\\
F_{k,i_1} &::=\{F_{k,i_1,i_2}\mid i_1+i_2\leq k, i_2\in\mathbb{N}\}\\
&\vdots\\
F_{k,i_1,\ldots,i_{d-2}} &::=\{r_{i_1,\ldots,i_{d-2},i_{d-1}}[i_d]\mid\\
&\qquad i_{d_1}+i_d=k-(i_1+\ldots+i_{d-2})\} \quad .
\end{aligned}$$

As an example, in Figure 4.6, $[p_{0,0}[1],p_{0,1}[0]]$, and $[p_{1,0}[0]]$ constitute plane D_1. The corresponding frontier is giving by $F_1=\{F_{1,0},F_{1,1}\}=\{\{r_{0,0}[1],r_{0,1}[0]\},\{r_{1,0}[0]\}\}$.

We define the *cuttable* property recursively, with the definition for the 2-dimensional case (c.f. Lemma 2) as base case. For indices $i_1,\ldots,i_l$ we also write $\bar{i}$. Two sub-planes $D_{k,\bar{i}}$, $D_{k',\bar{i}'}$ with $F_{k,\bar{i}}=F_{k',\bar{i}'}$ are *cuttable* if for every two $F_{k',\bar{i}',j'}\in F_{k',\bar{i}'}$ and $F_{k,\bar{i},j}\in F_{k,\bar{i}}$ with $F_{k',\bar{i}',j'}=F_{k,\bar{i},j}$, sub-frontier $F_{k,\bar{i},j}$ is associated with at least as many sub-sequences in $D_{k,\bar{i}}$ as $F_{k',\bar{i}'}$ is associated with in $D_{k',\bar{i}'}$, or at least with $|Q|$ many (where Q is the set of automata states). Furthermore, if $F_{k,\bar{i},j}$ is associated with $D_{k,\bar{i},j}$, and $F_{k',\bar{i}',j'}$ is associated with $D_{k',\bar{i}',j'}$, the corresponding sub-planes must be cuttable again.

We show how the *cut* operation can be extended by one dimension. A plane in the 3-dimensional comb is a sequence of sequences $D_{k,i}$ of nodes. Each sequence $D_{k,i}$ contains nodes that reside on paths branching from p_i. To cut plane $D_{k'}$ to D_k, we require that $F_k = F_{k'}$. Pick for each set $F_{k,i}$ an equal set $F_{k',i'}$ and cut the diagonal $D_{k',i'}$ to $D_{k,i}$, as described for the 2-dimensional case in the proof of Lemma 2. To preserve the relations of paths in the third dimension ($p_{i,j}$) with the paths in the first and second dimension (p and p_i), also replace each 2-dimensional sub-comb with origin at $p_i[k-i]$ with the sub-comb at $p_{i'}[k'-i']$; and the 3-dimensional sub-comb with origin at $p[k]$ with the sub-comb $p[k']$. Using the lifted definition of cuttable, it is also possible to define *preserving cuts* by first declaring a set of representative positions and then choosing the sets in such a way that no representative positions are deleted during the cut.

With the same arguments as in the 2-dimensional case and using the lifted (preserving) cut operation, we can create a 3-dimensional comb prefix $\mathcal{T}_{fin}$ of bounded size which can then be extended to a satisfying model $\tilde{\mathcal{T}}$. Note that the bound in the 3-dimensional case is exponentially larger than the bound in the 2-dimensional case due to the more complicated definition of cuttable. $\square$

We have, thus, proven that the existential fragment of HyperCTL* remains decidable. In the next section, we will consider quantifier alternations.

4.2.3 Fragments with a Quantifier Alternation

In this section, we will prove that a quantifier alternation of any kind leads to undecidability of HyperCTL*. HyperCTL* formulas in the general $\exists^*\forall^*$ fragment are undecidable. We prove this by spanning a comb-shaped tree with the $\exists\pi\Box\exists\pi'$ pattern. We then encode PCP into this tree with universal trace quantification.

Theorem 14. *The satisfiability problem for HyperCTL* formulas in the* $\exists^*\forall^*$ *fragment is undecidable.*

Proof. Let a PCP instance with $\Sigma = \{a_1, a_2, ..., a_n\}$ and two lists α and β be given. We choose our set of atomic propositions as follows: $AP ::= \{i\} \cup (\Sigma \cup \{\dot{a}_1, \dot{a}_2, ..., \dot{a}_n\} \cup \#)^2$, where we use the dot symbol to encode that a stone starts at this position of the trace. We write $\tilde{a}$ to denote either a or $\dot{a}$. The single input i is used to span the comb-shaped tree. We encode the PCP instance into a HyperCTL* formula that is satisfiable if and only if the PCP instance has a solution:

$$\begin{aligned}
&\exists\pi.\Box i && (4.1)\\
&\land(\Box\exists\pi'.\bigcirc\Box\neg i_{\pi'}) && (4.2)\\
&\land(\forall\pi''\forall\tilde{\pi}''.(i_{\pi''}\land\bigcirc\neg i_{\pi''})\rightarrow\bigcirc\varphi_{sol}(\pi'') && (4.3)\\
&\quad\land\varphi_{adjacent}(\pi'',\tilde{\pi}'')\rightarrow\varphi_{start}(\varphi_{stone\&shift}(\pi'',\bigcirc\tilde{\pi}''),\pi''))\ , && (4.4)
\end{aligned}$$

- Conjunct 4.1 defines the back of the comb, which is globally labeled with i.
- Conjunct 4.2 defines the tines of the comb, which are at the first position labeled with i, followed by globally $\neg i$.
- The first part of the last conjunct (4.3) requires that the solution of the PCP instance lives on the first tine of the comb.
- The second part of the last conjunct (4.4) requires that the next adjacent tine encodes the same trace as the previous tine but with the first stone from the PCP instance being removed as follows:
  - $\varphi_{start}(\varphi,\pi'') := i_{\pi''}\,\mathrm{U}(\varphi\land\Box\neg i_{\pi''})$ cuts of the irrelevant prefixes.
  - $\varphi_{adjacent}(\pi'',\tilde{\pi}'') := i_{\pi''}\land i_{\tilde{\pi}''}\,\mathrm{U}\,\neg i_{\pi''}\land\bigcirc\neg i_{\tilde{\pi}''}$ defines two adjacent tines.
  - $\varphi_{stone\&shift}(\pi'',\bigcirc\tilde{\pi}'')$ defines that the next adjacent tine removes a stone from the PCP instance as detailed outlined in the proofs in Chapter 3. The notation $\bigcirc\tilde{\pi}''$ denotes that the *stone&shift* encoding must be shifted by one position to match the adjacent tine.

□

We can, furthermore, follow that HyperCTL* inherits the undeciable $\forall^*\exists^*$ fragment of HyperLTL.

Corollary 7. *The satisfiability problem for HyperCTL* formulas in the* $\forall^*\exists^*$ *fragment is undecidable.*

Proof. HyperCTL* subsumes HyperLTL as a syntactic fragment. The $\forall^*\exists^*$ fragment of HyperLTL is known to be undecidable [70]. □

The results above are due to the indirect quantifier scopes of HyperCTL* [105] For example, the globally operator in the formula $\exists\pi.\Box(\exists\pi'.\forall\pi''.\varphi)$ spans a comb in such a way that $\forall\pi''$ indirectly refers to traces introduced by $\exists\pi'$. In the formula $\exists\pi.\Box(\exists\pi'.\bigcirc\forall\pi''.\varphi)$, the universal quantification only refers to the current trace π'. Formulas, which are in the indirect scope free $\exists^*\forall^*$ fragment however, can be decided by eliminating the universal quantification.

Theorem 15. *([105]) Satisfiability of the indirect scope free $\exists^*\forall^*$ fragment of HyperCTL* is decidable.*

We have thus identified the largest decidable fragment of HyperCTL*, which is the indirect scope free $\exists^*\forall^*$ fragment.

4.3 HyperCTL* Realizability

In this section, we briefly discuss the realizability problem of HyperCTL*. We conclude that the problem is undecidable in general and that HyperCTL* inherits the decidable fragments of HyperLTL.

Definition 15 (HyperCTL* Realizability). *A HyperCTL* formula φ over atomic propositions $AP = I \mathbin{\dot\cup} O$ is realizable if there is a strategy $f : (2^I)^* \rightarrow 2^O$ that satisfies φ.*

The linear $\forall^*_\pi$ fragment [74] for HyperLTL (and for HyperQPTL 3) can also be defined for HyperCTL*:

Definition 16. *Let $O = \{o_1, \ldots, o_n\}$. A HyperCTL* formula φ is called* linear *if for all $o_i \in O$ there is a $J_i \subseteq I$ such that $\varphi \wedge D_{I \mapsto O} \equiv collapse(\varphi) \wedge \bigwedge_{o_i \in O} D_{J_i \mapsto \{o_i\}}$ and $J_i \subseteq J_{i+1}$ for all $i \leq n$.*

We defined the collapsed formula of φ as $collapse(\varphi) ::= \forall \pi. Q^*_q. \psi[\pi_1 \mapsto \pi][\pi_2 \mapsto \pi] \ldots [\pi_n \mapsto \pi]$ where $\psi[\pi_i \mapsto \pi]$ replaces all occurrences of π_i in ψ with π. This means that every input-output dependency can be ordered linearly, i.e., we are restricted to linear architectures without information forks (see Figure 3.3). We know, however, that two universal quantifiers in a prefix (i.e. in HyperLTL) are enough to already encode the distributed synthesis problem [74]. HyperCTL* thus inherits the decidable fragments of HyperLTL, but remains undecidable in general.

Corollary 8. *HyperCTL* realizability inherits the decidable fragments of HyperLTL.*

Corollary 9. *HyperCTL* realizability is undecidable in general.*

4.4 Summary

We identified the decidability boundary of the satisfiability problem of HyperCTL*. We showed in this branching-time hierarchy of hyperlogics, that the $\exists^*$ fragment of HyperCTL* is decidable. This result enabled the development of HyperCTL* satisfiability solvers for the indirect scope free $\exists^*\forall^*$ HyperCTL* fragment [105]. Furthermore, we briefly concluded that the realizability problem of HyperCTL* is undecidable and that HyperCTL* inherits the decidable fragments of HyperLTL.

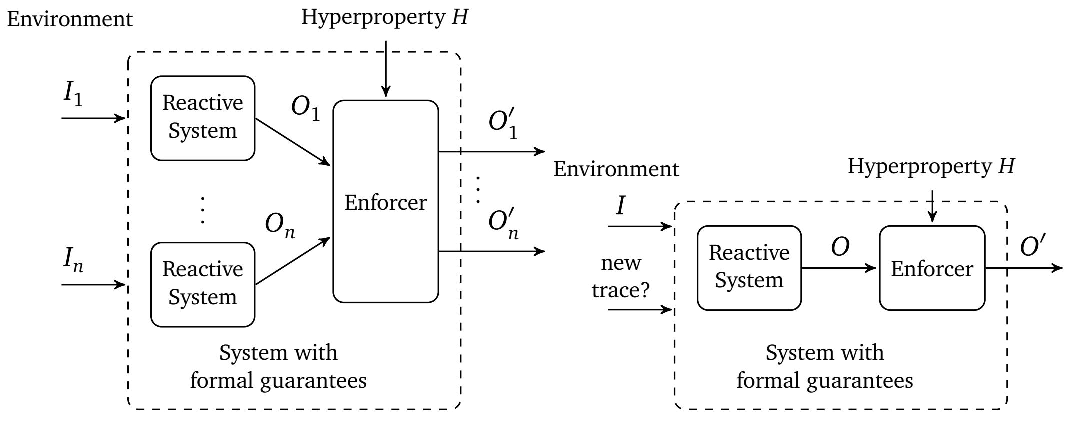
Chapter 5

Runtime Enforcement of Temporal Hyperproperties

Runtime enforcement combines the strengths of dynamic and static verification by monitoring the output of a running system and correcting it if it violates a given specification. Enforcement mechanisms thus provide formal guarantees for settings in which a system needs to be kept alive while also fulfilling critical properties. Runtime enforcement has been successfully applied in settings where specifications are given as trace properties [65, 63]. Privacy policies, for example, cannot be ensured by shutting down the system to prevent leakage: an attacker could gain information just from the fact that the execution stopped.

Our contribution in this chapter is two-fold. First, we show that hyperproperty enforcement of reactive systems needs to solve challenging variants of the synthesis problem. The concrete formulation depends on the given trace input model. We distinguish two input models 1) the parallel trace input model, where the number of traces is known a priori and traces are processed in parallel, and 2) the sequential trace input model, where traces are processed sequentially, and no a priori bound on the number of traces is known. Figure 5.1 repeats the visualization from the introduction of the general setting of reactive runtime enforcement in these input models. In the parallel trace input model, the enforcement mechanism observes n traces at the same time. This is the natural model if a system runs in secure multi-execution [48]. In the sequential trace input model, system runs are observed in sessions, i.e., one at a time. An additional input indicates that a new session (i.e., trace) starts. Instances of this model naturally appear, for example, in web-based applications.

Second, we describe enforcement mechanisms for a concrete specification language. In the previous chapters, we studied the theoretical boundaries of the satisfiability and realizability problem for highly expressive logics. In this chapter, we will move down the hierarchy of hyperlogics to HyperLTL, for which practical im-



(a) Parallel model. (b) Sequential model.

Figure 5.1: Runtime enforcement for a reactive system. In case the input-output-relation would violate the hyperproperty H, the enforcer corrects the output.

plementations exist. HyperLTL is still flexible enough to state different application-tailored specifications. Furthermore, the computational cost of the satisfiability and realizability problem is reasonable. We focus on universally quantified formulas, a fragment in which most of the enforceable hyperproperties naturally reside. For both trace input models, we develop enforcement mechanisms based on parity game solving. For the sequential model, we show that the problem is undecidable in general. However, we provide algorithms for the more straightforward case that the enforcer only guarantees a correct continuation for the current session. Furthermore, we describe an algorithm for the case that the specification describes a safety property. When it is not necessary for an enforcer to distinguish between inputs and outputs, we show that it is sufficient to solve a variation of the satisfiability problem at runtime. Our algorithms monitor for *losing* prefixes, i.e., so-far observed traces for which the system has no winning strategy against an adversarial environment. We ensure that our enforcement mechanisms are *sound* by detecting losing prefixes at the earliest possible point in time. Furthermore, they are *transparent*, i.e., non-losing prefixes are not altered.

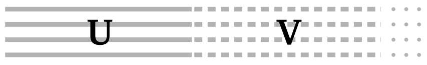
We accompany our findings with a prototype implementation for the parallel model and conduct two experiments: 1) we enforce symmetry in mutual exclusion algorithms, and 2) we enforce the information flow policy observational determinism. We will see that enforcing such complex HyperLTL specifications can scale to large traces once the initial parity game solving succeeds.

Results in this chapter are based on “Runtime Enforcement of Hyperproperties” [43], which was joint work with Norine Coenen, Bernd Finkbeiner, Jana Hofmann, and Yannick Schillo. The chapter is structured as follows. In Section 5.1, we develop a formal defintion of temporal hyperproperty enforcement. In Section 5.2, we give explicit enforcement algorithms for specifications given in universally quantified HyperLTL. We provide experimental results in Section 5.3 before concluding in Section 5.4.

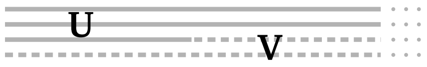
5.1 Hyperproperty Enforcement

In this section, we develop a formal definition of hyperproperty enforcement mechanisms for reactive systems modeled with two trace input models. To this end, we first formally describe reactive systems under the two trace input models by the *prefixes* they can produce. We proceed by developing the two basic requirements on enforcement mechanisms, *soundness* and *transparency* [63, 64], for our settings. Soundness is traditionally formulated as “the enforced system should be correct w.r.t. the specification”. Transparency (also known as precision [156]) states that “the system’s behavior should be modified minimally, i.e., the longest correct prefix should be preserved by the enforcement mechanism”. In the context of reactive systems, formal definitions for soundness and transparency need to be formulated in terms of strategies. They describe how the enforcement mechanism reacts to the inputs from the environment and outputs produced by the system. We, therefore, define soundness and transparency based on the notion of *losing prefixes* (i.e., prefixes for that no winning strategy exists) inspired by work on monitoring reactive systems [55]. We will see that the definition of losing prefixes depends heavily on the chosen trace input model. Primarily the sequential model defines an interesting new kind of synthesis problem, which varies significantly from the known HyperLTL synthesis problem.

As is common in the study of runtime techniques for reactive systems, we make the following reasonable assumptions. First, reactive systems are treated as *black boxes*, i.e., two reactive systems with the same observable input-output behavior are considered equal. Thus, enforcement mechanisms cannot base their decisions on implementation details. Second, w.l.o.g. and to simplify the presentation, we assume execution traces to have *infinite length*. Finite traces can always be interpreted as infinite traces, e.g., by adding end^{ω}. To reason about finite traces, on the other hand, definitions like the semantics of HyperLTL would need to accommodate many special cases like traces of different lengths. Lastly, we assume that control stays with the enforcer after a violation occurred instead of only correcting the error and handing control back to the system afterward. Since we aim to provide formal guarantees, these two problems are equivalent. To ensure that the correction adheres to the specification, the enforcer needs to guarantee that there is a strategy for reacting to future inputs.



(a) Prefix in the parallel model.



(b) Prefix in the sequential model.

Figure 5.2: Visualization of prefixes in trace input models.

5.1.1 Trace Input Models

We distinguish two *trace input models* [77], the parallel and the sequential model. The trace input models describe how a reactive system is employed and how its traces are obtained (see Figure 5.1). We formally define the input models by the prefixes they can produce. The definitions are visualized in Figure 5.2. In the parallel model, a fixed number of n systems are executed in parallel, producing n events at a time.

Definition 17 (Prefix in the Parallel Model)**.** *An n-tuple of finite traces $U = (u_1, \dots u_n) \in ((2^\Sigma)^*)^n$ is a prefix of $V = (v_1, \dots v_n) \in ((2^\Sigma)^{*/\omega})^n$ (written $U \preceq V$) in the parallel model with n traces iff each u_i is a prefix of v_i (also denoted by $u_i \preceq v_i$).*

The prefix definition models the allowed executions of a system under the parallel trace input model: If the system produces U and after a few more steps produces V, then $U \preceq V$. Note that the prefix definition is transitive: U can be a prefix of another prefix (then the traces in V are of finite length) or a prefix of infinite-length traces.

In the sequential model, the traces are produced one by one and there is no a-priori known bound on the number of traces.

Definition 18 (Prefix in the Sequential Model)**.** *Let $U = (u_1, \dots, u_n) \in ((2^\Sigma)^\omega)^*$ be a sequence of traces and $u \in (2^\Sigma)^*$ be a finite trace. Let furthermore $V = (v_1, \dots, v_n, \dots)$ be a (possibly infinite) sequence of traces with $v_i \in (2^\Sigma)^\omega$, and $v \in (2^\Sigma)^*$ be a finite trace. We call (U, u) a prefix of (V, v) (written $(U, u) \preceq (V, v)$) iff either 1) $U = V$ and $u \preceq v$ or 2) $V = u_1, \dots, u_n, v_{n+1}, \dots$ and $u \preceq v_{n+1}$.*

We additionally say that $(U, u) \preceq V$ if $(U, u) \preceq (V, \epsilon)$, where ϵ is the empty trace. To continue an existing prefix (U, u), the system either extends the started trace u or finishes u and continues with additional traces. Traces in U are of infinite length and describe finished sessions. This means that they cannot be modified after the start of a new session. Again, prefixes in this model are transitive and are also defined for infinite sets.

We defined prefixes tailored to the trace input models to precisely capture the influence of the models on the enforcement problem. Usually, a set of traces T is defined as prefix of a set of traces T' if and only if $\forall t \in T.\ \exists t' \in T'.\ t \preceq t'$ [41]. A prefix in the sequential model, however, *cannot* be captured by the traditional prefix definition, as it does not admit infinite traces in a prefix.

5.1.2 Losing Prefixes for Hyperproperties

Losing prefixes describe *when* an enforcer has to intervene based on possible strategies for future inputs. As we will see, the definition of losing prefixes, and thus the definition of the enforcement problem, differs significantly for both input models. For the rest of this section, let H denote an arbitrary hyperproperty.

We first define strategies for the parallel model with n parallel sessions. In the enforcement setting, a strategy receives a previously recorded prefix. Depending on that prefix, the enforcer's strategy might react differently to future inputs. We therefore define a *prefixed strategy* as a higher-order function $\sigma : ((2^\Sigma)^*)^n \to ((2^I)^*)^n \to (2^O)^n$ over $\Sigma = I \mathbin{\dot\cup} O$. The strategy first receives a prefix (produced by the system), then a sequence of inputs on all n traces, and reacts with an output for all traces. We define a losing prefix as follows.

Definition 19 (Losing Prefix in the Parallel Model)**.** *A strategy $\sigma(U)$ is losing for H with $U = (u_1, \dots, u_n) \in ((2^\Sigma)^*)^n$ if there are input sequences $(v_1, \dots, v_n) \in ((2^I)^\omega)^n$ such that the following set is not in H:*

$$\bigcup_{1 \leq i \leq n} \{u_i \cdot (v_i[0] \cup \sigma_U(\epsilon)(i)) \cdot (v_i[1] \cup \sigma_U(v_i[0])(i)) \cdot (v_i[2] \cup \sigma_U(v_i[0]v_i[1])(i)) \dots\},$$

where $\sigma_U = \sigma(U)$ and $\sigma_U(\cdot)(i)$ denotes the i-th output that σ produces.

We say that $\sigma(U)$ is winning if it is not losing. A prefix U is winning if there is a strategy σ such that $\sigma(U)$ is winning. Lastly, σ is winning if $\sigma(\epsilon)$ is winning and for all non-empty winning prefixes U, $\sigma(U)$ is winning. Similar to the parallel model, a prefixed strategy in the sequential model is a function $\sigma : ((2^\Sigma)^\omega)^* \times (2^\Sigma)^* \to (2^I)^* \to 2^O$ over $\Sigma = I \mathbin{\dot\cup} O$. The definition of a losing prefix is the following.

Definition 20 (Losing Prefix in the Sequential Model)**.** *In the sequential model, a strategy σ is losing with a prefix (U, u) for H, if there are input sequences $V = (v_0, v_1, \dots)$ with $v_i \in (2^I)^\omega$, such that the set $U \cup \{t_0, t_1, \dots\}$ is not in H, where $t_0, t_1, \dots$ are defined as follows:*

$$\begin{aligned}
t_0 &:= u \cdot (v_0[0] \cup \sigma(U,u)(\epsilon)) \cdot (v_0[1] \cup \sigma(U,u)(v_0[0])) \cdot \dots \\
t_1 &:= (v_1[0] \cup \sigma(U \cup \{t_0\}, \epsilon)(\epsilon)) \cdot (v_1[1] \cup \sigma(U \cup \{t_0\}, \epsilon)(v_1[0])) \cdot \dots \\
t_2 &:= (v_2[0] \cup \sigma(U \cup \{t_0, t_1\}, \epsilon)(\epsilon)) \cdot (v_2[1] \cup \sigma(U \cup \{t_0, t_1\}, \epsilon)(v_2[0])) \cdot \dots
\end{aligned}$$

Winning prefixes and strategies are defined analogously to the parallel model. The above definitions illustrate that enforcing hyperproperties in the sequential model defines an intriguing but complex problem. Strategies react to inputs based on the observed prefix. The *same* input sequence can therefore be answered differently in the first session and, say, in the third session. The enforcement problem thus not simply combines monitoring and synthesis but formulates a different kind of problem.

5.1.3 Enforcement Mechanisms

With the definitions of the previous sections, we adapt the notions of sound and transparent enforcement mechanisms to hyperproperties under the two trace input models. We define an enforcement mechanism *enf* for a hyperproperty H to be a computable function which transforms a black-box reactive system S with trace input model $\mathcal{M}$ into a reactive system $enf(S)$ with the same input model.

Definition 21 (Soundness). *enf is* sound *if for all reactive systems S and all input sequences in model $\mathcal{M}$, the set of traces produced by $enf(S)$ is in H.*

Definition 22 (Transparency). *enf is* transparent *if the following holds: Let U be a prefix producible by S with input sequence s_I. If U is winning, then for any prefix V producible by $enf(S)$ with input sequence s'_I where $s_I \preceq s'_I$, it holds that $U \preceq V$.*

We now have everything in place to define when a hyperproperty is enforcable for a given input model.

Definition 23 (Enforceable Hyperproperties). *A hyperproperty H is* enforceable *if there is a sound and transparent enforcement mechanism.*

In order to obtain a sound and transparent enforcement mechanism, we need to construct a winning strategy for H.

Proposition 1. *Let H be a hyperproperty and $\mathcal{M}$ be an input model. Assume that it is decidable whether a prefix U is losing in model $\mathcal{M}$ for H. Then there exists a sound and transparent enforcement mechanism enf for H iff there exists a winning strategy in $\mathcal{M}$ for H.*

Proof. For the first direction, let *enf* be given. We design a winning strategy σ as follows. Let a prefix U be given, which is either empty or winning. Note that we cannot exploit the fact that U is winning: We know that there is a strategy but we do not know how to compute it. Instead, let S be a reactive system which produces U. If U is empty, let S produce *false* in the first step. Compute $enf(S)$. Now let V be an input sequence suitable for $\mathcal{M}$ such that $U_{|I} = V$. Since *enf* is transparent, the outputs of $enf(S)$ in reaction to V produce U. Now, for any further inputs, σ reacts according to $enf(S)$. Since $enf(S)$ is sound, σ is winning.

For the second direction, let a reactive system S be given. Note that since σ is winning, $\sigma(\epsilon)$ is winning. We define $S' = enf(S)$ recursively as follows: Let U be the prefix computed so far. Invariantly, it is a winning prefix. Let input i be given (the type of i depends on the given trace input model). Compute $o = S(i)$. Decide whether U together with (i, o) is winning. If it is winning, go on with the next input. If it is losing, then return $\sigma(U)(i)$ instead of o and use $\sigma(U)$ for all future inputs. The described enforcement mechanism *enf* is sound as either the traces producible by S are in H or at some point $\sigma(U)$ for a winning prefix U takes over control. It is transparent, as the output of S is not alternated as long as the prefix is winning. □

The above proposition describes how to construct enforcement algorithms: We need to solve the synthesis problem posed by the respective trace input model. However, we have to restrict ourselves to properties that can be monitored for losing prefixes. This is only natural: for example, the property expressed by the HyperLTL formula $\exists \pi. \square a_\pi$ can in general not be enforced since it contains a hyperliveness [41] aspect: There is always the possibility for the required trace π to occur in a future session (c.f. the definition of monitorable hyperproperties in [7, 77]). We therefore describe algorithms for HyperLTL specifications from the universal fragment $\forall \pi_1. \dots \forall \pi_k. \varphi$ of HyperLTL. Additionally, we assume that the specification describes a property whose counterexamples have losing prefixes.

Before jumping to concrete algorithms, we describe two example scenarios of hyperproperty enforcement with different trace input models.

In the first example, we consider fairness in contract signing. Contract signing protocols let multiple parties negotiate a contract. In this setting, fairness requires that in every situation where Bob can obtain Alice's signature, Alice must also be able to obtain Bob's signature. Due to the asymmetric nature of contract signing protocols (one party has to commit first), fairness is difficult to achieve (see, e.g., [158]). Many protocols rely on a trusted third party (TTP) to guarantee fairness. The TTP may negotiate multiple contracts in parallel sessions. The natural trace input model is therefore the parallel model. Fairness forbids the existence of two traces π and π' that have the same prefix of inputs, followed in π by Bob requesting (R^B) and receiving the signed contract (S^B), and in π' by Alice requesting (R^A), but *not* receiving the signed contract ($\neg S^A$):

$$\forall \pi. \forall \pi'. \neg((\bigwedge_{i \in I}(i_\pi \leftrightarrow i_{\pi'}))\ \mathcal{U}\ (R^B_\pi \wedge R^A_{\pi'} \wedge \bigcirc(S^B_\pi \wedge \neg S^A_{\pi'})))\ .$$

In the second example we consider privacy in fitness trackers. Wearables track a wide range of extremely private health data which can leak an astonishing amount of insight into your health. For instance, it has been found that observing out-of-the-ordinary heart rate values correlates with diseases like the common cold or even Lyme disease [142]. Consider the following setting. A fitness tracker continuously collects data that is stored locally on the user's device. Additionally, the data is synced with an external cloud. While locally stored data should be left untouched, uploaded data has to be enforced to comply with information flow policies. Each day, a new stream of data is uploaded, hence the sequential trace input model would be appropriate. Comparing newer streams with older streams allows for the detection of anomalies. We formalize an exemplary property of this scenario in HyperLTL. Let *HR* be the set of possible heart rates. Let furthermore *active* denote whether the user is currently exercising. Then the following property ensures that unusually high heart rate values are not reported to the cloud:

$$\forall \pi . \forall \pi' . \square (\mathit{active}_\pi \leftrightarrow \mathit{active}_{\pi'} \rightarrow \bigwedge_{r \in HR} (r_\pi \leftrightarrow r_{\pi'})) \ .$$

5.2 Enforcement Algorithms for HyperLTL Specifications

For both trace input models, we present sound and transparent enforcement algorithms for universal HyperLTL formulas defining hyperproperties with losing prefixes. First, we construct an algorithm for the parallel input model based on parity game solving. For the sequential trace input model, we show that the problem is undecidable in the most general case. We proceed to provide an algorithm that only finishes the remainder of the current session. This simplifies the problem because the existence of a correct future session is not guaranteed. For this setting, we then present a simpler algorithm that is restricted to safety specifications.

5.2.1 Parallel Trace Input Model

In short, we proceed as follows: First, since we know the number of traces, we can translate the HyperLTL formula to an equivalent LTL formula. For that formula, we construct a realizability monitor based on the LTL monitor described in [55]. The monitor is a parity game, which we use to detect minimal losing prefixes and to provide a valid continuation for the original HyperLTL formula.

Assume that the input model contains n traces. Let a HyperLTL formula $\forall \pi_1 \ldots \forall \pi_k . \varphi$ over $\Sigma = I \,\dot\cup\, O$ be given, where φ is quantifier free. We construct an LTL formula φ^n_{LTL} over $\Sigma' = \{a_i \mid a \in \Sigma, 1 \leq i \leq n\}$ as follows:

$$\varphi^n_{\mathrm{LTL}} := \bigwedge_{i_1,\ldots,i_k \in [1,n]} \varphi[\forall a \in AP : a_{\pi_1} \mapsto a_{i_1}, \ldots, a_{\pi_k} \mapsto a_{i_k}] \ .$$

The formula φ^n_{LTL} enumerates all possible combinations to choose k traces – one for each quantifier – from the set of n traces in the model. We use the notation $\varphi[\forall a \in AP : a_{\pi_1} \mapsto a_{i_1}, \ldots, a_{\pi_k} \mapsto a_{i_k}]$ to indicate that in φ, atomic propositions with trace variables are replaced by atomic propositions indexed with one of the n traces. We define $I' = \{a_i \mid a \in I, 1 \leq i \leq n\}$ and O' analogously. Since n is known upfront, we only write φ_{LTL}.

Our algorithm exploits that for every LTL formula φ, there exists an equivalent parity game $\mathcal{G}_\varphi$ such that φ is realizable iff player P_0 is winning in the initial state with strategy σ_0 [60] (see Chapter 2). For a finite trace u, φ is realizable with prefix u

Algorithm 1 HyperLTL enforcement algorithm for the parallel input model.

```
 1: procedure INITIALIZE($\psi, n$)
 2:     $\varphi_{\text{LTL}}$ := toLTL($\psi, n$);
 3:     (game, $q_0$) := toParity($\varphi_{\text{LTL}}$);
 4:     winR := SolveParity(game);
 5:     if $q_0 \notin$ winR then
 6:         raise error;
 7:     return(game, winR, $q_0$);

 8: procedure ENFORCE(game, lastq)
 9:     sig := GetStrat(game, lastq);
10:     while true do
11:         o := sig(lastq);
12:         lastq:= Move(game, lastq, o);
13:         output(o);
14:         i := getNextInput( );
15:         $i_{\text{LTL}}$ := toLTL(i);
16:         lastq := Move(game, lastq, $i_{\text{LTL}}$);

17: procedure MONITOR(game, winR, q)
18:     lastq := q;
19:     while true do
20:         o := getNextOutput( );
21:         $o_{\text{LTL}}$ := toLTL(o);
22:         q := Move(game, lastq, $o_{\text{LTL}}$);
23:         if q $\notin$ winR then
24:             return(game, lastq);
25:         i := getNextInput( );
26:         $i_{\text{LTL}}$ := toLTL(i);
27:         q := Move(game, q, $i_{\text{LTL}}$);
28:         lastq := q;
```

iff the play induced by u ends in a state q that is in the winning region of player P_0. The algorithm to enforce the HyperLTL formula calls the following three procedures – depicted in Algorithm 1 – in the appropriate order.

Initialize: Construct φ_{LTL} and the induced parity game $\mathcal{G}_\varphi$. Solve the game $\mathcal{G}_\varphi$, i.e. compute the winning region for player P_0. If the initial state $q_0 \in V_0$ is losing, raise an error. Otherwise start monitoring in the initial state.

Monitor: Assume the game is currently in state $q \in V_0$. Get the next outputs $(o_1, \dots, o_n) \in O^n$ produced by the n traces of the system and translate them to $o_{\text{LTL}} \subseteq O'$ by subscripting them as described for formula φ_{LTL}. Move with o_{LTL} to the next state. This state is in V_1. Check if the reached state is still in the winning region. If not, it is a losing state, so we do not approve the system's output but let the enforcer take over and call ENFORCE on the last state. If the state is still in the winning region, we process the next inputs $(i_1, \dots, i_n)$, translate them to i_{LTL}, and move with i_{LTL} to the next state in the game, again in V_0. While the game does not leave the winning region, the property is still realizable and the enforcer does not need to intervene.

Enforce: By construction, we start with a state $q \in V_0$ that is in the winning region, i.e., there is a positional winning strategy $\sigma : V_0 \to 2^{O'}$ for player P_0. Using this strategy, we output $\sigma(q)$ and continue with the next incoming input i_{LTL} to the next state in V_0. Continue with this strategy for any incoming input.

Correctness and Complexity. By construction, since we never leave the winning region, the enforced system fulfills the specification and the enforcer is sound. It is also transparent: As long as the prefix produced by the system is not losing, the enforcer does not intervene. The algorithm has triple exponential complexity in the number of traces n: The size of φ_{LTL} is exponential in n and constructing the parity game is doubly exponential in the size of φ_{LTL} [60]. Solving the parity game only requires quasi-polynomial time [33, 161]. Note, however, that all of the above steps are part of the initialization. At runtime, the algorithm only follows the game arena. If the enforcer is only supposed to correct a single output and afterwards hand back control to the system, the algorithm can be adapted accordingly.

Non-reactive Setting

In general, we assume systems to be reactive. There are however situations where distinguishing between inputs and outputs is not necessary. This is the case, for example, when monitoring the outgoing stream of a device to a cloud like in the fitness-tracker example given in the introduction. We show that in such a situation, it is not necessary to build a parity game. Indeed, existing algorithms for HyperLTL monitoring [77] can be easily extended to also enforce a hyperproperty. In [77], monitoring of HyperLTL formulas is studied with respect to both trace input models. Their algorithm for the parallel model already recognizes minimal bad prefixes. We can therefore concentrate on how to provide a valid continuation once a minimal bad prefix is detected.

Let a HyperLTL formula φ from the universal HyperLTL fragment, and the number of traces n be given. Assume that a HyperLTL monitor for the parallel trace input model is running. Each event e^i consists of a tuple $(e^i_1, \ldots, e^i_n) \in (2^\Sigma)^n$. As long as the monitor does not raise an alarm, new events are forwarded to the monitor. If the monitor raises an alarm at position $i+1$, we encode the observed traces up to position i into a HyperLTL formula, together with the formula we want to enforce. The resulting formula is the following:

$$\begin{aligned} \psi := {} & \exists \pi_1 \ldots . \exists \pi_n . \varphi \wedge \\ & \bigwedge_{1 \leq k \leq n} \bigwedge_{0 \leq j \leq i} \bigcirc^j \Big(\bigwedge_{a \in e^j_k} a_{\pi_k} \wedge \bigwedge_{a \notin e^j_k} \neg a_{\pi_k} \Big) \ . \end{aligned}$$

We use $\bigcirc^j$ as an abbreviation for j consecutive $\bigcirc$. The first conjunct (φ) is the original universal HyperLTL formula. The second conjunct ensures that a satisfying trace set contains at least n traces with the prefixes seen so far. The resulting formula ψ is an $\exists^*\forall^*$ HyperLTL formula. It is forwarded to a satisfiability solver for temporal hyperproperties (e.g., EAHyper [75]) which, if the result is SAT, returns at least n traces $(t_1, \ldots, t_n)$ such that the traces $(t_1[i+1\ldots], \ldots, t_n[i+1\ldots])$ can be used to continue the observed prefix and enforce φ. Note that we did not ask for *exactly* n traces. Universally quantified HyperLTL formulas are downwards closed, i.e., any additional traces returned by the SAT solver can just be ignored. Also note that the result must be SAT by construction. If the result was UNSAT, then the already observed prefix up to position i would form a minimal bad prefix. Then, the monitor would have raised an alarm at position i already.

Compared to the parity-game-based algorithms for reactive systems, we do not solve a synthesis problem, but a satisfiability problem. Satisfiability of $\exists^*\forall^*$ HyperLTL formulas is solvable in single-exponential space (see Chapter 2), compared to the triple exponential time complexity for the reactive case. However, the parity game can be computed beforehand, while the satisfiability problem needs to be solved at runtime (since we need to encode the observed traces). The evaluation of these two approaches thus depends on the specific situation at hand, i.e., on whether the parity game can be computed at all and what computational overhead at runtime is considered acceptable.

This algorithm takes over the control of the enforced system when a bad prefix was about to occur. The enforcer then fully determines how the system executions are continued in the future. If we only want to avoid the violation of the specification in the current situation but give back the control to the system afterwards, this algorithm can be adapted accordingly: Given a monitor alarm at position $i+1$, we solve the above formula ψ and use $(t_1[i+1], \ldots, t_n[i+1])$ as the event for position $i+1$. After that, we go back to monitoring the system for the next bad prefix. In order to give a correct enforcement, even only for one position, one still has to compute a solution for all coming positions. Otherwise, there might be no valid continuation in the future.

5.2.2 Sequential Trace Input Model

Deciding whether a prefix is losing in the sequential model is harder than in the parallel model. In the sequential model, strategies are defined w.r.t. the traces seen so far – they incrementally upgrade their knowledge with every new trace. In general, the question whether there exists a sound and transparent enforcement mechanism for universal HyperLTL specifications is undecidable.

Theorem 16. *In the sequential model, it is undecidable whether a HyperLTL formula φ from the universal fragment is enforceable.*

Proof. We encode the classic realizability problem of universal HyperLTL, which is undecidable [73], into the sequential model enforcement problem for universal HyperLTL. The HyperLTL realizability problem asks if there exists a strategy $\sigma\colon (2^I)^* \to 2^O$ such that the set of traces constructed from every possible input sequence satisfies the formula φ, i.e. whether $\{(w[0] \cup \sigma(\epsilon)) \cdot (w[1] \cup \sigma(w[0])) \cdot (w[2] \cup \sigma(w[0..1])) \cdot \ldots \mid w \in (2^I)^\omega\}, \emptyset \models \varphi$. Let a universal HyperLTL formula φ over $\Sigma = I \,\dot\cup\, O$ be given. We construct the universal HyperLTL formula

$$\psi := \varphi \wedge \forall \pi. \forall \pi'. (\bigwedge_{o \in O} o_\pi \leftrightarrow o_{\pi'}) \mathrel{\mathrm{W}} (\bigvee_{i \in I} i_\pi \not\leftrightarrow i_{\pi'}) \ .$$

Formula ψ requires the strategy to choose the same outputs as long as the inputs are the same. The choice of the strategy must therefore be independent of earlier sessions, i.e., $\sigma(U, \epsilon)(s_I) = \sigma(U', \epsilon)(s_I)$ for all sets of traces U, U' and input sequences s_I. Any trace set that fulfills ψ can therefore be arranged in a traditional HyperLTL strategy tree branching on the inputs and labeling the nodes with the outputs. Assume the enforcer has to take over control after the first event when enforcing ψ. Thus, there is a sound and transparent enforcement mechanism for ψ iff φ is realizable. $\square$

Finishing the Current Session

As the general problem is undecidable, we study the problem where the enforcer takes over control only for the rest of the current session. For the next session, the existence of a solution is not guaranteed. This approach is especially reasonable if we are confident that errors occur only sporadically. We adapt the algorithm presented for the parallel model. Let a HyperLTL formula $\forall \pi_1. \ldots \forall \pi_k. \varphi$ over $\Sigma = I \,\dot\cup\, O$ be given, where φ is quantifier free. As for the parallel model, we translate the formula into an LTL formula φ^n_{LTL}. We first do so for the first session with $n = 1$. We construct and solve the parity game for that formula, and use it to monitor the incoming events and to enforce the rest of the session if necessary. For the next session, we construct φ^n_{LTL} for $n = 2$ and add an additional conjunct encoding the observed trace t_1. The resulting formula induces a parity game that monitors and enforces the second trace. Like this, we can always enforce the current trace in relation to all traces seen so far. Algorithm 2 depicts the algorithm calling similar procedures as in Algorithm 1 (for which we therefore do not give any pseudo code). INITIALIZE' is already given an LTL formula and, therefore, does not translate its input to LTL. MONITOR' returns a tuple including the reason for its termination ('ok' when the trace finished and 'losing' when a losing prefix was detected). Additionally, the monitor returns the trace seen so far (not including the event that led to a losing prefix), which will be added to φ_{traces}. ENFORCE' enforces the rest of the session and afterwards returns the produced trace, which is then encoded in the LTL formula (TOLTL(t)).

Correctness and Complexity. Soundness and transparency follow from the fact that for the n-th session, the algorithm reduces the problem to the parallel setting with n traces, with the first $n-1$ traces being fixed and encoded into the LTL formula φ^n_{LTL}.

Algorithm 2 HyperLTL enforcement algorithm for the sequential trace input model.

```
1: procedure EnforceSequential($\psi$)
2:     n := 1;
3:     $\varphi_{\text{traces}}$ := true;
4:     while true do
5:         $\psi_{\text{curr}}$ := toLTL($\psi$, n) $\wedge$ $\varphi_{\text{traces}}$;
6:         (game, winR, $q_0$) := Initialize'($\psi_{\text{curr}}$);
7:         res := Monitor'(game, winR, $q_0$);
8:         if res == ('ok', t) then
9:             $\varphi_{\text{traces}}$ := $\varphi_{\text{traces}}$ $\wedge$ toLTL(t);
10:        else if res = ('losing', t, (game, lastq)) then
11:            t' := Enforce'(game, lastq);
12:            $\varphi_{\text{traces}}$ := $\varphi_{\text{traces}}$ $\wedge$ toLTL(t $\cdot$ t');
13:        n++;
```

We construct a new parity game from φ^n_{LTL} after each finished session. The algorithm is thus of non-elementary complexity.

Safety Specifications

If we restrict ourselves to formulas $\psi = \forall\pi_1 \ldots \forall\pi_k.\, \varphi$, where φ is a *safety* formula, we can improve the complexity of the algorithm. Note, however, that not every property with losing prefixes is a safety property: for the formula $\forall\pi.\square(o_\pi \rightarrow \diamond i_\pi)$ with $o \in O$ and $i \in I$, any prefix with o set at some point is losing. However, the formula does not belong to the safety fragment. Given a safety formula φ, we can translate it to a safety game [132] instead of a parity game. The LTL formula we create with every new trace is built incrementally, i.e., with every finished trace we only ever add new conjuncts. With safety games, we can thus recycle the winning region from the game of the previous trace. The algorithm proceeds as follows. 1) Translate φ into an LTL formula φ^n_{LTL} for $n = 1$. 2) Build the safety game $\mathcal{G}^1_{\varphi_{\text{LTL}}}$ for φ^1_{LTL} and solve it. Monitor the incoming events of trace t_1 as before. Enforce the rest of the trace if necessary. 3) Once the session is terminated, generate the LTL formula $\varphi^2_{\text{LTL}} = \varphi^1_{\text{LTL}} \wedge \varphi^2_{\text{diff}}$. As φ^2_{LTL} is a conjunction of the old formula and a new conjunct φ^2_{diff}, we only need to generate the safety game $\mathcal{G}^2_{\text{diff}}$ and then build the product of $\mathcal{G}^2_{\text{diff}}$ with the winning region of $\mathcal{G}^1_{\varphi_{\text{LTL}}}$. We solve the resulting game and monitor (and potentially enforce) as before. The algorithm incrementally refines the safety game and enforces the rest of a session if needed. The construction recycles parts of the game computed for the previous session. We thus avoid the costly translation to a parity game for every new session. While constructing the safety game from the LTL specification has still doubly exponential complexity [132], solving safety games can be done in linear time [16].

Non-reactive Setting

Similarly to the parallel input model, we can encode enforcement in the sequential model as a HyperLTL SAT problem, for the special case where the enforcer does not need to distinguish between inputs and outputs. Again, we employ existing monitoring tools and show how to provide an enforcement in case an error is observed. Let a HyperLTL monitor for the sequential model be given, e.g., the one from [77]. Note that in the sequential setting, finished sessions naturally produce finite-length traces. We assume that the resulting traces are extended to infinite ones by appending the trace $\{end\}^\omega$. Assume a violation of the property is detected in session n at position $c+1$. Furthermore, for each finished session $k < n$, let the trace length be i_k.

We remove the last step that caused the violation (since the monitor monitors for minimal prefixes) and solve the satisfiability problem of the following formula:

$$\psi := \exists \pi_1 \ldots \exists \pi_n . \bigwedge_{1 \leq k < n} \bigcirc^{i_k} \square end_{\pi_k} \wedge \bigwedge_{0 \leq j \leq i_k} \bigcirc^{j} \Big(\bigwedge_{a \in e_k^j} a_{\pi_k} \wedge \bigwedge_{a \notin e_k^j} \neg a_{\pi_k} \Big)$$
$$\bigwedge_{0 \leq j < c} \bigcirc^{j} \Big(\bigwedge_{a \in e_n^j} a_{\pi_n} \wedge \bigwedge_{a \notin e_n^j} \neg a_{\pi_n} \Big) \ .$$

The first conjunct encodes all finished sessions. The second conjunct encodes all events seen so far for the current session n. Any trace set satisfying $\psi \wedge \varphi$ provides a valid continuation for trace n, finishing the current session. Furthermore, for future traces, the enforcer can choose any trace from that trace set (since the specification is universal and the enforcer has control over all atomic propositions).

5.3 Experimental Evaluation

We implemented the algorithm for the parallel trace input model in a prototype tool, which is written in Rust. We use Strix [152] for the generation of the parity game. We determine the winning region and the positional strategies of the game with PG-Solver [91]. All experiments ran on an Intel Xeon CPU E3-1240 v5 3.50 GHz, with 8 GB memory running Debian 10.6. We evaluate our prototype with two experiments. In the first, we enforce a non-trivial formulation of fairness in a mutual exclusion protocol. In the second, we enforce the information flow policy *observational determinism* on randomly generated traces.

	random traces				symmetric traces			
$\|t\|$	avg	min	max	#enforced	avg	min	max	#enforced
500	0.003	0.003	0.003	0	0.013	0.008	0.020	10
1000	0.005	0.005	0.005	0	0.024	0.015	0.039	10
5.000	0.026	0.024	0.045	0	0.078	0.065	0.097	10
10.000	0.049	0.047	0.064	0	0.153	0.129	0.178	10

Table 5.1: Enforcing symmetry in the Bakery protocol on pairs of traces. Times are given in seconds.

5.3.1 Enforcing Symmetry in Mutual Exclusion Algorithms

Mutual exclusion algorithms like Lamport's bakery protocol ensure that multiple threads can safely access a shared resource. To ensure fair access to the resource, we want the protocol to be symmetric, i.e., for any two traces where the roles of the two processes are swapped, the grants are swapped accordingly. Since symmetry requires the comparison of two traces, it is a hyperproperty. For our experiment, we used a Verilog implementation of the Bakery protocol [135], which has been proven to violate the following symmetry formulation [82]:

$$\forall \pi. \forall \pi'. (pc(0)_\pi = pc(1)_{\pi'} \wedge pc(1)_\pi = pc(0)_{\pi'}) \mathbin{\mathrm{W}} (pause_\pi = pause_{\pi'} \wedge \\ \texttt{sym}(sel_\pi, sel_{\pi'}) \wedge \texttt{sym}(break_\pi, break_{\pi'}) \wedge sel_\pi < 3 \wedge sel_{\pi'} < 3) \enspace .$$

The specification states that for any two traces, the program counters need to be symmetrical in the two processes as long as the processes are scheduled (*select*) and ties are broken (*break*) symmetrically. Both *pause* and *sel* < 3 handle further implementation details. The AIGER [18] translation [82] of the protocol has 5 inputs and 46 outputs. To enforce the above formula, only 10 of the outputs are relevant. We enforced symmetry of the bakery protocol on simulated pairs of traces produced by the protocol. Table 5.1 shows our results for different trace lengths and trace generation techniques. We report the average runtime over 10 runs as well as minimal and maximal times along with the number of times the enforcer needed to intervene. The symmetry assumptions are fairly specific and are unlikely to be reproduced by random input simulation. In a second experiment, we therefore generate pairs of symmetric traces. Here, the enforcer had to intervene every time, which produces only a small overhead.

The required game was constructed and solved in 313 seconds. For sets of more than two traces, the construction of the parity game did not return within two hours. The case study shows that the tool performs without significant overhead at runtime and can easily handle very long traces. The bottleneck is the initial parity game construction and solving.

benchmark size				init	0.5% bit flip probability				1% bit flip probability			
#i	#o	#t	|t|		avg	min	max	#enf	avg	min	max	#enf
1	1	3	10K	0.5	0.014	0.013	0.017	60	0.013	0.013	0.015	60
		8	10K	65.7	0.524	0.517	0.625	99	0.524	0.517	0.588	97
2	2	4	10K	0.87	0.025	0.024	0.030	73	0.032	0.031	0.043	77
		5	10K	21.19	0.038	0.037	0.041	90	0.038	0.037	0.042	86
3	3	2	10K	0.6	0.019	0.018	0.022	47	0.023	0.023	0.025	54
		4	5K	132.8	0.022	0.021	0.026	77	0.021	0.021	0.021	71
		4	10K		0.038	0.036	0.056	77	0.037	0.037	0.042	77
4	4	3	1K	43.9	0.010	0.008	0.015	71	0.009	0.008	0.018	68
		3	5K		0.023	0.021	0.033	72	0.022	0.021	0.025	78
		3	10K		0.038	0.037	0.050	76	0.038	0.037	0.041	75

Table 5.2: Enforcing observational determinism. Times are given in seconds.

5.3.2 Enforcing Observational Determinism

In our second experiment, we enforced observational determinism, given as the HyperLTL formula $\forall\pi.\forall\pi'.(o_\pi \leftrightarrow o_{\pi'})\mathcal{W}(i_\pi \not\leftrightarrow i_{\pi'})$. The formula states that for any two execution traces, the observable outputs have to agree as long as the observable inputs agree. Observational determinism is a prototypical information-flow policy used in many experiments and case studies for HyperLTL (e.g. [73, 82, 23]). We generated traces using the following scalable generation scheme: At each position, each input and output bit is flipped with a certain probability (0.5% or 1%). This results in instances where observational determinism randomly breaks. Table 7.1 shows our results. Each line corresponds to 100 randomly generated instances of the given size (number of inputs/outputs and traces, and length of the sessions). We report the initialization time that is needed to generate and solve the parity game. Furthermore, we report average, minimal, and maximal enforcement time as well as the number of instances where the enforcer intervened. Times are reported in seconds. The bottleneck is the time needed to construct and solve the parity game. At runtime, which is the crucial aspect, the enforcer performs efficiently. The higher bit flip probability did not lead to more enforcements: For traces of length 10000, the probability of intervention is relatively high already at a bit flip probability of 0.5%.

5.4 Summary

We studied the runtime enforcement problem for hyperproperties. Depending on the trace input model, we showed that the enforcement problem boils down to detecting losing prefixes and solving a custom synthesis problem. For both input models, we provided enforcement algorithms for specifications given in the universally quantified fragment of the temporal hyperlogic HyperLTL. While the problem for the sequential trace input model is in general undecidable, we showed that enforcing HyperLTL specifications becomes decidable under the reasonable restriction to only finish the current session. For the parallel model, we provided an enforcement mechanism based on parity game solving. Our prototype tool implements this algorithm for the parallel model. We conducted experiments on two case studies enforcing complex HyperLTL specifications for reactive systems with the parallel model. Our results show that once the initial parity game solving succeeds, our approach has only little overhead at runtime and scales to long traces.

Part II

Deep Learning Methods for Temporal Reasoning

Chapter 6

Teaching Temporal Logics to Neural Networks

Machine learning has revolutionized several areas of computer science, such as image recognition [107], face recognition [201], translation [222], and board games [155, 193]. For complex tasks that involve symbolic reasoning, however, deep learning techniques are still considered as insufficient. Applications of deep learning in logical reasoning problems have therefore focused on sub-problems within larger logical frameworks, such as computing heuristics in solvers [136, 12, 188] or predicting individual proof steps [146, 94, 13, 112]. Recently, however, the assumption that deep learning is not yet ready to tackle hard logical questions was drawn into question: [134] demonstrated that Transformer models [214] perform surprisingly well on symbolic integration, [172] demonstrated that self-supervised training leads to mathematical reasoning abilities, and [28] demonstrated that large-enough language models learn basic arithmetic despite being trained on mostly natural language sources. This poses the question if challenging logical problems in verification that are thought to require symbolic reasoning lend themselves to a direct learning approach. We thus consider linear-time temporal logic (LTL) [165], which is widely used in the academic verification community [53, 139, 52, 175, 185, 139, 138, 186] and is the basis for industrial hardware specification languages like the IEEE standard PSL [114] or more expressive hyperlogics like HyperLTL [40]. LTL specifies infinite sequences and is typically used to describe system behaviors (see Chapter 2 for details). Logical methods of more hyperlogics can often be reduced to the corresponding LTL problems, such as the satisfiability problem of the $\exists^*\forall^*$ fragment of HyperLTL [70]. Successfully applying deep neural networks to logical methods of temporal logics *end-to-end* would immediately yield reliable heuristics, as the predictions of the neural network can be checked.

In this chapter, we apply a direct learning approach to the fundamental problem of LTL to find a satisfying trace to a formula. In applications, solutions to LTL formu-

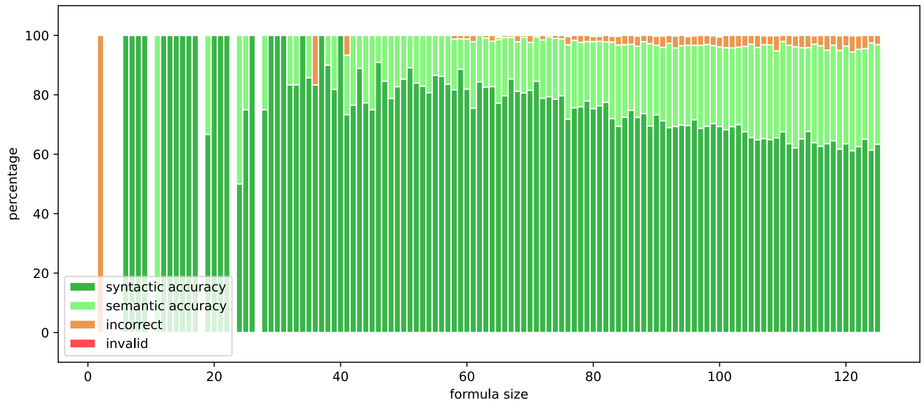


Figure 6.1: Performance of our best model trained on practical pattern formulas (*LTLPattern126*). The x-axis shows the formula size. Syntactic accuracy, i.e., where the Transformer agrees with the generator are displayed in dark green. Instances where the Transformer deviates from the generators output but still provides correct output are displayed in light green; incorrect predictions in orange.

las can represent (counter) examples for a specified system behavior, and over the last decades, generations of advanced algorithms have been developed to solve this question automatically. We start from the standard benchmark distribution of LTL formulas, consisting of conjunctions of patterns typically encountered in practice [53]. We then use classical algorithms, notably `spot` [52], that implement a competitive classical algorithm, to generate solutions to formulas from this distribution and train a Transformer model to predict these solutions directly.

Relatively small Transformers perform very well on this task and we predict correct solutions to 96.8% of the formulas from a held-out test set (see Figure 6.1). Impressive enough, Transformers hold up pretty well and predict correct solutions in 83% of the cases, even when we focus on formulas on which `spot` timed out. This means that, already today, direct machine learning approaches may be useful to augment classical algorithms in logical reasoning tasks.

We also study two generalization properties of the Transformer architecture, important to logical problems: We present detailed analyses on the generalization to *longer formulas*. It turns out that transformers trained with tree-positional encodings [192] generalize to much longer formulas than they were trained on, while Transformers trained with the standard positional encoding (as expected) do not generalize to longer formulas. The second generalization property studied here is the question whether Transformers learn to imitate the generator of the training data, or whether they learn to solve the formulas according to the semantics of the logics. This question arises because for most formulas there are many possible satisfying traces. In Figure 6.1 we highlight the fact that our models often predicted traces that satisfy the formulas, but predict different traces than the one found by the classical algorithm with which we generated the data. Especially when testing the models out-of-distribution we observed that almost no predicted trace equals the solution proposed by the classical solver.

To demonstrate that these generalization behaviors are not specific to the benchmark set of LTL formulas, we also present experimental results on random LTL formulas. Further, we exclude that `spot`, the tool with which we generate example traces, is responsible for these behaviors, by repeating the experiments on propositional formulas for which we generate the solutions by SAT solvers.

Results in this chapter are based on “Teaching Temporal Logics to Neural Networks” [102], which was joint work with Frederik Schmitt, Jens U. Kreber, Markus N. Rabe, and Bernd Finkbeiner. The chapter is structured as follows. We give an overview over the Transformer architecture in Section 6.1. We describe the problem definitions and present our data generation in Section 6.2. Our experimental setup is described in Section 6.3 and our findings in Section 6.4, before concluding in Section 6.5.

6.1 The Transformer

The Transformer [214] is a deep neural network architecture initially proposed for solving natural language processing tasks. They have become the state-of-the-art architecture for many natural language processing tasks, such as translation or summarization, replacing, e.g., recurrent neural networks (RNNs) such as long short-term memories (LSTMs) [109]. Transformers are designed to handle sequences of input elements by computing hidden embeddings for each element in parallel.

6.1.1 Architecture Overview

Transformers make use of the so-called *attention* mechanism that enables the Transformer to relate arbitrary input elements to each other and to process the input sequence all at once. We will explain this mechanism in detail further below. There are several benefits to this approach: First, the training can be massively parallelized, and can thus effectively exploit modern hardware. Second, the processing of the input data in the neural network is very flexible, i.e., the input elements do not have to be processed one after another. Third, the number of steps information has to flow through the neural network is significantly decreased, alleviating problems with computing gradients through many operations.

There are publicly available implementations of the Transformer, e.g., [27, 213], which can handle a wide range of tasks. This makes this approach highly accessible

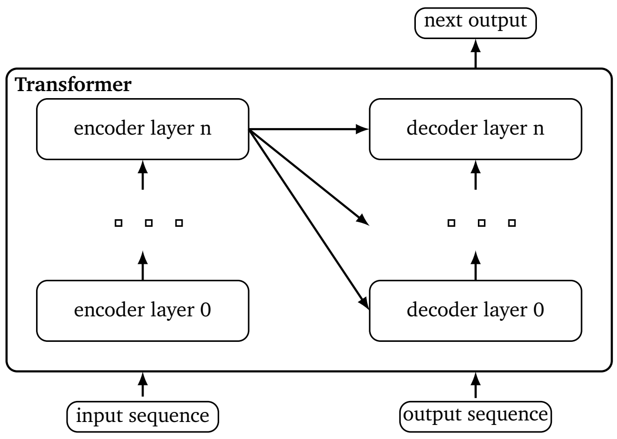


Figure 6.2: A high-level overview of the Transformer architecture: The input sequence is processed in one go by multiple encoding layers. The intermediate result is then given to each decoder layer. The decoder takes the already computed output and computes the next output step-by-step.

for users outside the machine learning domain. We begin by describing an overview of the architecture before zooming into the details of the architecture, i.e., the encoder, decoder, and attention mechanisms.

A Transformer follows a basic encoder-decoder structure (see Figure 6.2). The encoder constructs a hidden embedding z_i for each input embedding x_i of the input sequence $x = (x_0, \ldots, x_n)$ in one go. An embedding is a mapping from plain input, for example words or characters, to a high dimensional vector, for which learning algorithms and toolkits exists, e.g., word2vec [153]. Given the encoders output $z = (z_0, \ldots, z_k)$, the decoder generates a sequence of output embeddings $y = (y_0, \ldots, y_m)$ step-by-step. Note that the length of the input and output sequences does not have to be the same. Since the transformer architecture contains no recurrence nor any convolution, a positional encoding is added to the input and output embeddings that allows to distinguish between different orderings.

6.1.2 Encoder

The encoder consists of multiple layers where each layer is composed of the following two components: a so-called self-attention mechanism and a fully connected feed-forward neural network (see Figure 6.3). Each component is followed by a layer-normalization [11] step, which significantly reduces the training time by normalizing the activities of the neurons. The feed-forward neural network *FFNN* consists of two linear transformations with a ReLU activation in between, i.e., $\mathit{FFNN}(x) = \mathit{max}(0, xW_1 + b_1)W_2 + b_2$. Instead of maintaining a hidden state, e.g. in a recurrent

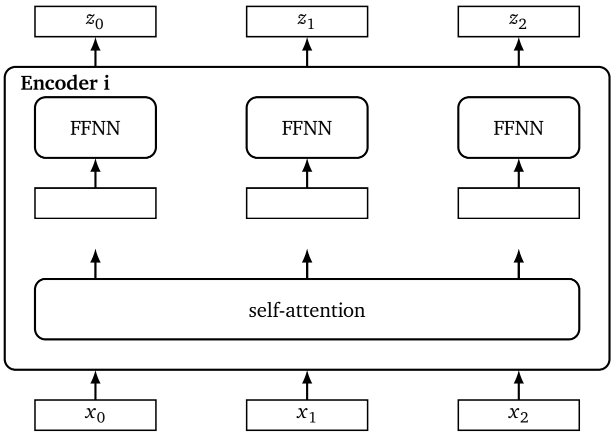


Figure 6.3: One encoder layer of the Transformer: Every embedding x_i is processed by a self-attention layer. The result is then processed by the same feed-forward neural network. Each embedding z_i is then either given to the next encoder layer or, if it is the last encoder layer, to the decoder.

neural network architecture, the self-attention mechanism allows the neural network to incorporate the hidden embedding of other important input elements into the hidden embedding of the current element under consideration.

We begin by describing the notion of attention intuitively. Consider a Transformer trained for translation and the input sentence “The animal didn’t cross the street because it was too tired”. Figure 6.4 (left) depicts the attention that the neural network pays to the other words in the sentence when encoding the word “it”. The most attention is focused on the phrase “The animal”. Computing the attention is parallelized in multiple attention-heads that potentially pay attention to different parts of the input. Figure 6.4 (right) shows one attention head during the encoding of the formula $(a \,\mathrm{U}\, b) \wedge (a \,\mathrm{U} \neg b)$. When encoding the second until-operator this particular attention head pays very close attention to the b of the first until-operator. Intuitively, it pays attention to the fact that the second until-operator has to take the first conjunct into account as well. This means the Transformer cannot simply satisfy the formula by outputting $\neg b$ and b on the first position of the trace as this would lead to a contradiction. As we will see in our experimental results, the transformer constructs the following trace instead: $a \wedge \neg b$; b ; true^ω, where ; denotes the beginning of the next position. I.e., it delays the satisfaction of the first until to the second position.

We describe the attention mechanism in more technical detail. To compute the hidden embedding for the i-th character of the formula we fist add the aforementioned positional encoding to the input embeddings x_i. Then, the self-attention is computed as follows. For each input embedding x_i, we compute 1) a query vector q_i, 2) a key vector k_i, and 3) a value vector v_i by multiplying x_i with weight matrices

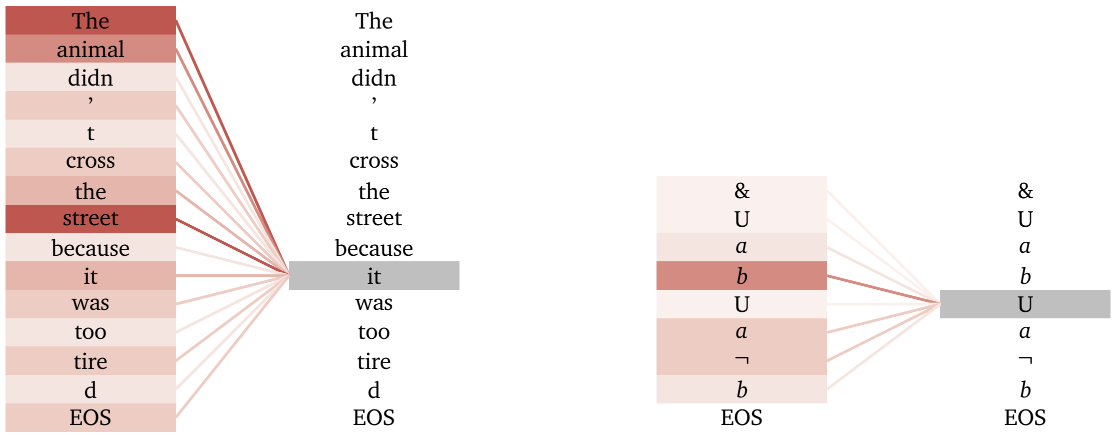


Figure 6.4: Self attention of the input of a Transformer that solves a translation task (left) [8] and a Transformer that understands an LTL formula (right). Stronger colored edges correspond to higher attention values.

W_k, W_v, and W_q, which are learned during the training process.

The main idea of the self-attention mechanism is to compute a score for each pair (x_i, x_j) representing which positions in the sequence should be considered the most when computing the embedding of x_i. In our visualizations, for example in Figure 6.4, stronger colored edges correspond to a higher attention value [216]. This mechanism is especially suited for our LTL trace generation problem as the whole context of the formula has impact on the choice of the decoder.

This is implemented by the so-called Scaled Dot-Product Attention: For example, consider a self-attention computation of $x = (x_0, x_1, x_2)$. To compute the hidden embedding of x_0 we first take the dot products of the query vector q_0 and key vectors k_0, k_1, and k_2. Intuitively, the query vector asks for a "selection" of different keys that it wants to know more about. Those scores are then divided by a constant $\sqrt{d_k}$, where d_k is the key dimension, to obtain more stable gradients. Taking the softmax results in three attention scores s_0, s_1, and s_3, which are the attentions for (x_0, x_0), (x_0, x_1), and (x_0, x_2) respectively. Note that each s_i is between 0 and 1 and $s_0 + s_1 + s_2 = 1$. The hidden embedding of x_0 is then obtained by the linear combination of v_0, v_1, and v_2 where v_i is scaled with score s_i. This maps the queried keys to their values, where keys with higher attention scores contribute more to the embedding. Intuitively, this mechanism can be seen as an indexing scheme over a multi-dimensional vector space.

The embeddings can be calculated all at once using matrix operations [214]. Let Q, K, V be the matrices obtained by multiplying the input vector X consisting of all x_i with the weight matrices W_k, W_v, and W_q:

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad .$$

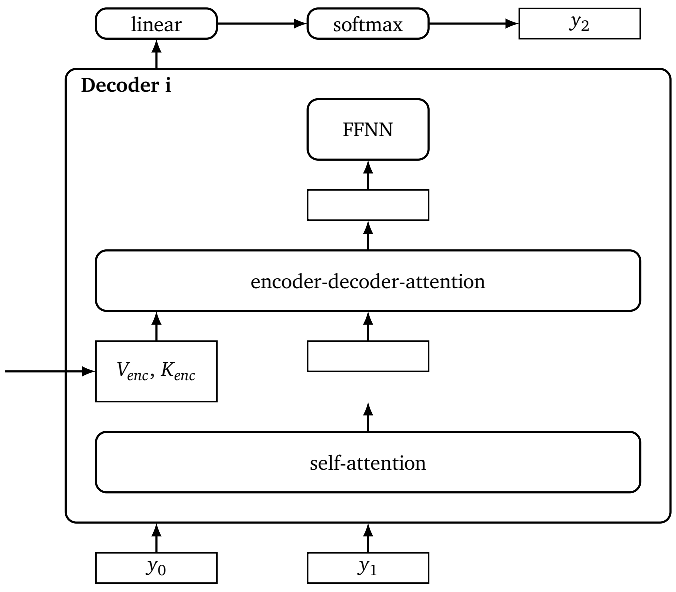


Figure 6.5: One decoder layer of the Transformer: Every already decoded output y_i is processed by a self-attention layer. Then, the encoder-decoder attention between the input embeddings $V_{enc} = (z_0, \ldots, z_k)$, given from the encoder, and the current output embedding is computed. If it is the last decoder layer, the next output element is chosen by a linearization and softmax operation.

6.1.3 Decoder

In contrast to the encoder, which processes the input sequence at once, the output sequences are computed step-by-step until the end of string (<EOS>) element is chosen (see Figure 6.5). The decoder is again a layered architecture. Each layer consists, in addition to a self-attention layer which processes already decoded elements and feed-forward neural network, of a layer that computes the attention between the output and input sequences. This is the key concept of the Transformer architecture: In contrast to the self-attention mechanism, the query Q_{dec} comes from the decoder asking for the values V_{enc} of the keys K_{enc} it is the most interested in while decoding a position of the output sequence, i.e., the $Attention(Q_{dec}, K_{enc}, V_{enc})$ is computed. Figure 6.6 shows an encoder-decoder attention head between the input formula $(a \, \mathrm{U} \, b) \wedge (a \, \mathrm{U} \neg b)$ and the symbolic trace $a \wedge \neg b \ ; \ b \ ; \ true^{\omega}$. This head focuses on matching the atomic proposition b of the second until operator to the first position of the trace and, thus, distinguishes between the same atomic propositions in the formula. This concludes our short explanation of the Transformer architecture. We describe our datasets and their generation in the next section.

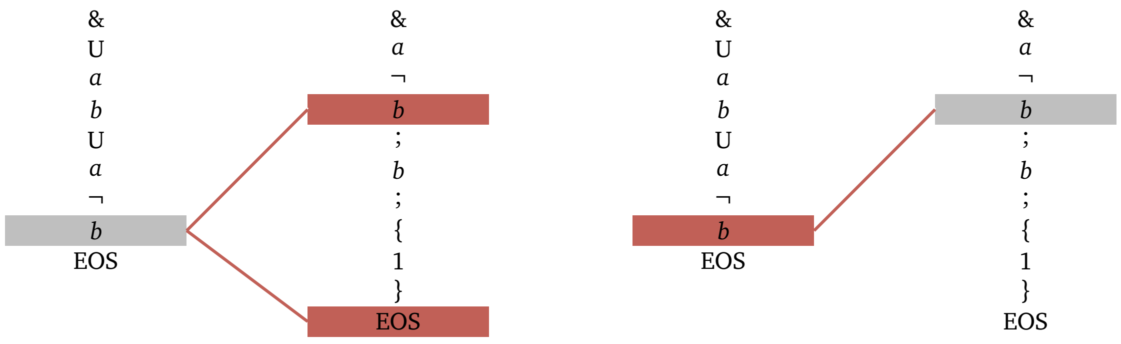


Figure 6.6: Attention values of one Encoder-decoder attention head for b between the formula $(a \, \mathrm{U} \, b) \wedge (a \, \mathrm{U} \, \neg b)$ and the output trace $a \wedge \neg b \; ; \; b \; ; \mathit{true}^{\omega}$ (and vice versa).

6.2 Datasets

To demonstrate the generalization properties of the Transformer on logical tasks, we generated several datasets in three different fashions. We will describe the underlying logical problems and our data generation in the following.

6.2.1 Trace Generation for Linear-time Temporal Logic

We consider infinite sequences, that are finitely represented in the form of a "lasso" uv^{ω}, where u, called prefix, and v, called period, are finite sequences of propositional formulas. We call such sequences *(symbolic) traces*. For example, the symbolic trace $(a \wedge b)^{\omega}$ defines the infinite sequence where a and b evaluate to true on every position. Symbolic traces allow us to underspecify propositions when they do not matter. For example, the LTL formula $\bigcirc\bigcirc\square a$ is satisfied by the symbolic trace: *true true* $(a)^{\omega}$, which allow for any combination of propositions on the first two positions.

Our datasets consist of pairs of satisfiable LTL formulas and satisfying symbolic traces generated with tools and automata constructions from the `spot` framework [52]. We use a compact syntax for ultimately periodic symbolic traces: Each position in the trace is separated by the delimiter ";". True and False are represented by "1" and "0", respectively. The beginning of the period v is signaled by the character "{" and analogously its end by "}". For example, the ultimately periodic symbolic trace denoted by $a; a; a; \{b\}$ describes all infinite traces where a holds on the first three positions followed by an infinite period of b's.

Given a satisfiable LTL formula φ, our trace generator constructs a Büchi automaton A_{φ} that accepts exactly the language defined by the LTL formula, i.e., $\mathcal{L}(A_{\varphi}) = \mathcal{L}(\varphi)$. From this automaton, we construct an arbitrary accepted symbolic trace, by searching for an accepting run in A_{φ}.

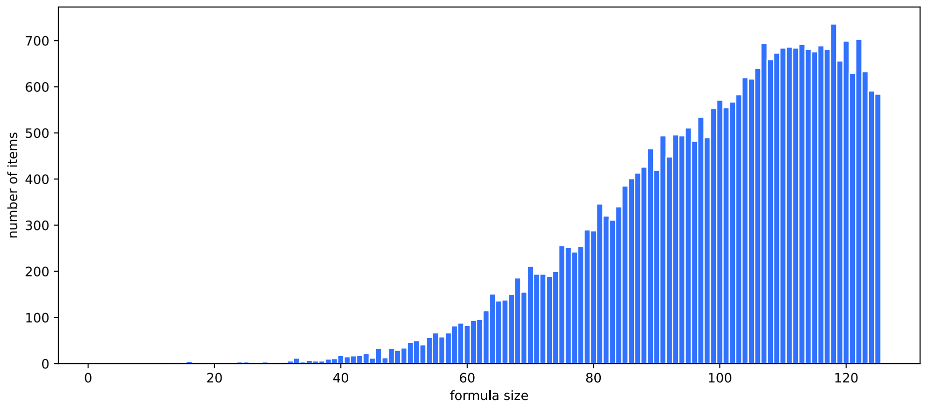


Figure 6.7: Size distributions in the *LTLPattern126* test set: on the x-axis is the size of the formulas; on the y-axis the number of formulas.

Specification Pattern

Our main dataset is constructed from formulas following 55 LTL specification patterns identified by the literature [53]. For example, the arbiter property $(\Diamond p_0) \rightarrow (p_1 \, \mathrm{U} \, p_0)$, stating that if p_0 is scheduled at some point in time, p_1 is scheduled until this point. The largest specification pattern is of size 40 consisting of 6 atomic propositions. It has been shown that conjunctions of such patterns are challenging for LTL satisfiability tools that rely on classical methods, such as automata constructions [139]. They start coming to their limits when more than 8 pattern formulas are conjoined. We decided to build our dataset in a similar way from these patterns only to allow for a better comparison.

We conjoined random specification patterns with randomly chosen variables (from a supply of 6 variables) until one of the following four conditions are met: 1) the formula size succeeds 126, 2) more than 8 formulas would be conjoined, 3) our automaton-based generator timed out ($> 1s$) while computing the solution trace, or 4) the formula would become unsatisfiable. In total, we generated 1664487 formula-trace pairs in 24 hours on 20 CPUs. While generating, approximately 41% of the instances ran into the first termination condition, 21% into the second, 37% into the third and 1% into the fourth. We split this set into an 80% training set, a 10% validation set, and a 10% test set. The size distribution of the dataset can be found in Figure 6.7.

For studying how the Transformer performs on longer specification patterns, we accumulated pattern formulas where `spot` timed out ($> 60s$) while searching for a satisfying trace. We call this dataset *LTLUnsolved254*. We capped the maximum length at 254, which is twice as large as the formulas the model saw during training.

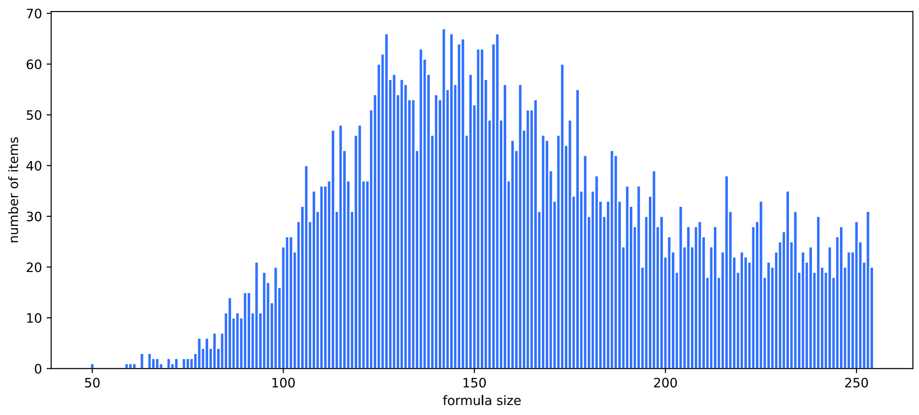


Figure 6.8: Size distributions in the *LTLUnsolved254* test set: on the x-axis is the size of the formulas; on the y-axis the number of formulas.

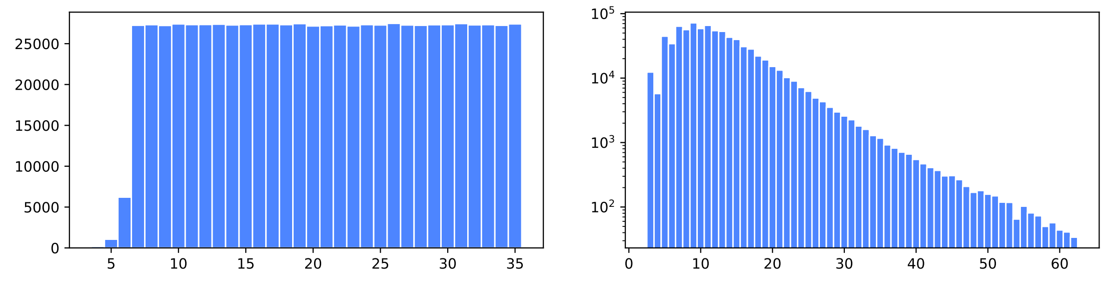
The size distribution of the generated formulas can be found in Figure 6.8.

In the following table, we illustrate the complexity of our training dataset with two examples from the above described set *LTLPattern*126, where the subsequent number of the notation of our datasets denotes the maximum size of a formula's syntax tree. The first line shows the LTL formula and the symbolic trace in mathematical notation. The second line shows the input and output representation of the Transformer (in Polish notation):

LTL formula	satisfying symbolic trace
$\Box(a \rightarrow \Diamond d) \land \neg f \,\mathrm{W}\, f \,\mathrm{W}\, \neg f \,\mathrm{W}\, f \,\mathrm{W}\, \Box \neg f$ $\land(\Diamond c \rightarrow \neg c \,\mathrm{U}\,(c \land \neg b \,\mathrm{W}\, b \,\mathrm{W}\, \neg b \,\mathrm{W}\, b \,\mathrm{W}\, \Box \neg b))$	$(\neg a \land \neg c \land \neg f \lor \neg c \land d \land \neg f)^{\omega}$
`&&G>aFdW!fWfW!fWfG!f>FcU!c&cW!bWbW!bWbG!b`	`{!a&!c&!f|!c&d&!f}`
$\Box(b \land \neg a \land \Diamond a \rightarrow c \,\mathrm{U}\, a) \land \Box(a \rightarrow \Box c) \land (\Diamond b \rightarrow \neg b$ $\mathrm{U}(b \land \neg f \,\mathrm{W}\, f \,\mathrm{W}\, \neg f \,\mathrm{W}\, f \,\mathrm{W}\, \Box \neg f)) \land (\Diamond a \rightarrow (c \land \bigcirc(\neg a \,\mathrm{U}\, e)$ $\rightarrow \bigcirc(\neg a \,\mathrm{U}\,(e \land \Diamond f))) \,\mathrm{U}\, a) \land \Diamond c \land \Box(a \Box \Diamond e \rightarrow \neg(\neg e \land f \land \bigcirc$ $(\neg e \,\mathrm{U}\,(\neg e \land d))) \,\mathrm{U}\,(e \lor c)) \land (\Box \neg a \lor \Diamond(a \land \neg f \,\mathrm{W}\, d)) \land \Box(e \rightarrow \Box \neg c)$	$(\neg a \land b \land \neg c \land \neg e \land f)(\neg a \land \neg c$ $\land \neg e \land \neg f)(\neg a \land \neg c \land \neg e \land f)$ $(\neg a \land c \land \neg e \land \neg f)(\neg a \land \neg e \land \neg f)^{\omega}$
`&&&&&&&&G>&&b!aFaUcaG>aGc>FbU!b&bW!fWfW!fW` `fG!f>FaU>&cXU!aeXU!a&eFfaFcG>&aFeU!&` `&!efXU!e&!ed|ec|G!aF&aW!fdG>eG!c`	`&&&&!ab!c!ef;&&&!a!c!e!f;` `&&&!a!c!ef;&&&!ac!e!f` `;{&&!a!e!f}`

Random Formulas

To show that the generalization properties of the Transformer are not specific to our data generation, we also generated a dataset of random formulas. Our dataset of random formulas consist of 1 million generated formulas and their solutions, i.e., a satisfying symbolic trace. The number of different propositions is fixed to 5. Each dataset is split into a training set of 800K formulas, a validation set of 100K formulas, and a test set of 100K formulas. All datasets are uniformly distributed in size,



(a) Formula distribution by size. (b) Trace distribution by size.

Figure 6.9: Size distributions in the *LTLRandom*35 training set: on the x-axis is the size of the formulas/traces; on the y-axis the number of formulas/traces.

apart from the lower-sized end due to the limited number of unique small formulas. The formula and trace distribution of the dataset *LTLRandom*35 can be found in Figure 6.9. Note that we filtered out examples with traces larger than 62 (less than 0.05% of the original set).

To generate the formulas, we used the `randltl` tool of the `spot` framework, which builds unique formulas in a specified size interval, following a supplied node probability distribution. During the building process, the actual distribution occasionally differs from the given distribution in order to meet the size constraints, e.g., by masking out all binary operators. The distribution between all k-ary nodes always remains the same. To furthermore achieve a (quasi) uniform distribution in size, we subsequently filtered the generated formulas. Our node distribution puts equal weight on all operators $\neg, \wedge, \bigcirc$ and U. Constants `True` and `False` are allowed with 2.5 times less probability than propositions.

In the following, we give three random examples from *LTLRandom*35 training set. The first line shows the LTL formula and the symbolic trace in mathematical notation. The second line shows the syntactic representation (in Polish notation):

LTL formula	satisfying symbolic trace
$\bigcirc((d \,\mathrm{U}\, c) \,\mathrm{U} \bigcirc\bigcirc d) \wedge \bigcirc(b \wedge \neg(\neg d \,\mathrm{U}\, c))$ `&XUUdcXXdX&b!U!dc`	$\mathit{true}\ (b \wedge \neg c \wedge \neg d)\ (\neg c \wedge d)\ d\ (\mathit{true})^{\omega}$ `1;&&b!c!d;&!cd;d;{1}`
$\neg\bigcirc((\bigcirc e \wedge (\mathit{true} \,\mathrm{U}\, b) \wedge \bigcirc c) \,\mathrm{U}\, c)$ `!XU&&XeU1bXcc`	$\mathit{true}\ (\neg b \wedge \neg c)\ (\neg b)^{\omega}$ `1;&!b!c;{!b}`
$\bigcirc\neg((\neg c \wedge d) \,\mathrm{U} \bigcirc d)$ `X!U&!cdXd`	$\mathit{true}\ (c \vee \neg d)\ (\neg d)\ (\mathit{true})^{\omega}$ `1;|c!d;!d;{1}`

6.2.2 Assignment Generation for Propositional Logic

To show that the generalization of the Transformer to the semantics of logics is not a unique attribute of LTL, we also generated a dataset for propositional logic. A propositional formula consists of Boolean operators $\wedge$ (and), $\vee$ (or), $\neg$ (not), and variables also called literals or propositions. We consider the derived operators $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$ (implication), $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$ (equivalence), and $\varphi_1 \oplus \varphi_2 \equiv \neg(\varphi_1 \leftrightarrow \varphi_2)$ (xor). Given a propositional Boolean formula φ, the satisfiability problem asks if there exists a Boolean assignment $\Pi : \mathcal{V} \mapsto \mathbb{B}$ for every literal in φ such that φ evaluates to *true*. For example, consider the following propositional formula, given in conjunctive normal form (CNF): $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$. A possible satisfying assignment for this formula would be $\{(x_1, \mathit{true}), (x_2, \mathit{false}), (x_3, \mathit{true})\}$. We allow a satisfying assignment to be *partial*, i.e., if the truth value of a propositions can be arbitrary, it will be omitted. For example, $\{(x_1, \mathit{true}), (x_3, \mathit{true})\}$ would be a satisfying partial assignment for the formula above. We define a *minimal unsatisfiable core* of an unsatisfiable formula φ, given in CNF, as an unsatisfiable subset of clauses φ_{core} of φ, such that every proper subset of clauses of φ_{core} is still satisfiable.

We, again, generated 1 million random formulas. For the generation of propositional formulas, the specified node distribution puts equal weight on $\wedge$, $\vee$, and $\neg$ operators and half as much weight on the derived operators $\leftrightarrow$ and $\oplus$ individually. In contrast to previous work [189], which is restricted to formulas in CNF, we allow an arbitrary formula structure and derived operators.

A satisfying assignment is represented as an alternating sequence of propositions and truth values, given as 0 and 1. The sequence $a0b1c0$, for example, represents the partial assignment $\{(a, \mathit{false}), (b, \mathit{true}), (c, \mathit{false})\}$, meaning that the truth values of propositions d and e can be chosen arbitrarily (note that we allow five propositions). We used pyaiger [215], which builds on Glucose 4 [10] as its underlying SAT solver. We construct the partial assignments with a standard method in SAT solving: We query the SAT solver for a minimal unsatisfiable core of the *negation* of the formula. To give the interested reader an idea of the level of difficulty of the dataset, the following table shows three random examples from our training set *PropRandom*35. The first line shows the formula and the assignment in mathematical notation. The second line shows the syntactic representation (in Polish notation):

propositional formula	satisfying partial assignment
$((d \wedge \neg e) \wedge (\neg a \vee \neg e)) \leftrightarrow ((\neg \oplus (\neg b \leftrightarrow \neg e))$	$\{(a,0),(b,0),(c,1),(d,1),(e,0)\}$
$\vee((e \oplus (b \wedge d)) \oplus \neg(\neg c \vee (\neg a \leftrightarrow e))))$	
`<->&&d!e|!a!e|xor!b<->!b!exorxore&bd!|!c<->!ae`	`a0b0c1d1e0`
$(c \vee e) \vee (\neg a \leftrightarrow \neg b)$	$\{(c,1)\}$
`||ce<->!a!b`	`c1`
$\neg((b \vee e) \oplus ((\neg a \vee (\neg d \leftrightarrow \neg e))$	$\{(d,1),(e,1)\}$
$\vee(\neg b \vee (((\neg a \wedge b) \wedge \neg b) \wedge d))))$	
`!xor!be||!a<->!d!e!|!b&&&!ab!b!d`	`d1e1`

To test the Transformer on even more challenging formulas, we constructed a dataset of CNF formulas using the generation script of [189] from their publicly available implementation. A random CNF formula is built by adding clauses until the addition of a further clause would lead to an unsatisfiable formula. We used the parameters $p_{geo} = 0.9$ and $p_{k2} = 0.75$ to generate formulas that contain up to 15 variables and have a maximum size of 250. We call this dataset *PropCNF*250.

6.3 Experimental Setup

We have implemented the Transformer architecture [214].[1] Our implementation processes the input and output sequences token-by-token. We trained on a single GPU (NVIDIA P100 or V100). All training has been done with a dropout rate of 0.1 and early stopping on the validation set. Note that the embedding size will automatically be floored to be divisible by the number of attention heads. The training of the best models took up to 50 hours. For the output decoding, we utilized a beam search [222], with a beam size of 3 and an α of 1.

Since the solution of a logical formula is not necessarily unique, we use two different measures of accuracy to evaluate the generalization to the semantics of the logics: we distinguish between the *syntactic* accuracy, i.e., the percentage where the Transformers prediction syntactically matches the output of our generator and the *semantic* accuracy, i.e., the percentage where the Transformer produced a different solution. We also differentiate between incorrect predictions and syntactically invalid outputs which, in fact, happens only in 0.1% of the cases in *LTLUnsolved254*.

In general, our best performing models used 8 layers, 8 attention heads, and a fully connected layer (FC) size of 1024. We used a batch size of 400 and trained for 450K steps (130 epochs) for our specification pattern dataset, and a batch size of 768 and trained for 50K steps (48 epochs) for our random formula dataset.

6.4 Experimental Results

In this section, we describe our experimental results. First, we show that a Transformer can indeed solve the task of providing a solution, i.e., a trace for a linear-time temporal logical (LTL) formula. For this, we describe the results from training on the dataset *LTLPattern126* of specification patterns that are commonly used in the context of verification. Furthermore, we provide training details, i.e., a hyperparameter analysis of models trained on *LTLRandom35* and the evolution of the syntactic and semantic accuracy during training. Secondly, we show two generalization properties that the Transformer evinces on logic reasoning tasks: 1) the generalization to larger formulas (even so large that our data generator timed out) and 2) the generalization

[1]The code, our data sets, and data generators are available at `https://github.com/reactive-systems/deepltl` and are part of the Python library ML2 (`https://github.com/reactive-systems/ml2`).

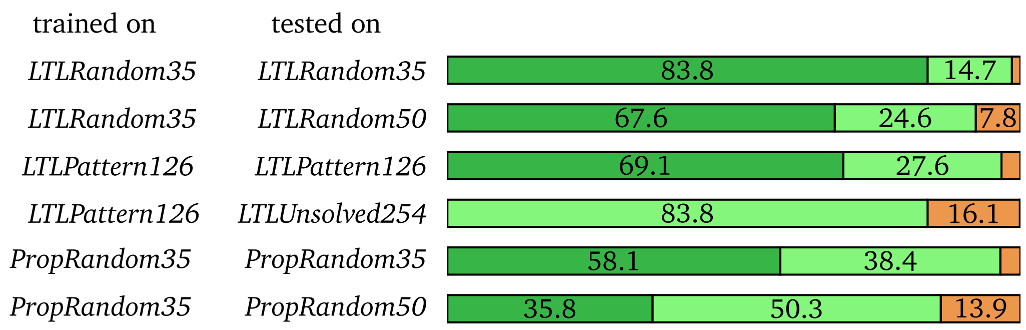


Figure 6.10: Overview of our main experimental results: the performance of our best performing models on our different datasets. The percentage of a dark green bar refers to the syntactic accuracy, the percentage of a light green bar to the semantic accuracy without the syntactic accuracy, and the incorrect predictions are visualized in orange.

to the semantics of the logic. We strengthen this observation by considering a different dataset of random LTL formulas. Thirdly, we provide results for a model trained on a different logic and with a different data generator. We thereby demonstrate that the generalization behaviors of the Transformer are not specific to LTL and the LTL solver implemented with spot that we used to generate the data. An overview of our training results is displayed in Figure 6.10.

6.4.1 Solving Linear-time Temporal Logical Formulas

We trained a Transformer on specification of our data set *LTLPattern126*. Figure 6.1 in the introduction displays the performance of our best model on this dataset. We observed a syntactic accuracy of 69.1% and a semantic accuracy of 96.8%. With this experiment we can already deduce that it seems easier for the Transformer to learn the underlying semantics of LTL than to learn the particularities of the generator. Further we can see that as the formula length grows, the syntactic accuracy begins to drop. However, that drop is much smaller in the semantic accuracy—the model still mostly predicts correct traces for long formulas.

As a challenging benchmark, we tested our best performing model on *LTLUnsolved254*. It predicted correct solutions in 83% of the cases, taking on average 15s on a single CPU. The syntactic accuracy is 0% as there was no output produced by `spot` within the timeout. The results of the experiments are visualized in Figure 6.11. Note that this does not mean that our Transformer models necessarily outperform classical algorithms across the board. However, since *verifying* solutions to LTL formulas is much easier than *finding* solutions (AC^1(logDCFL) [130] vs

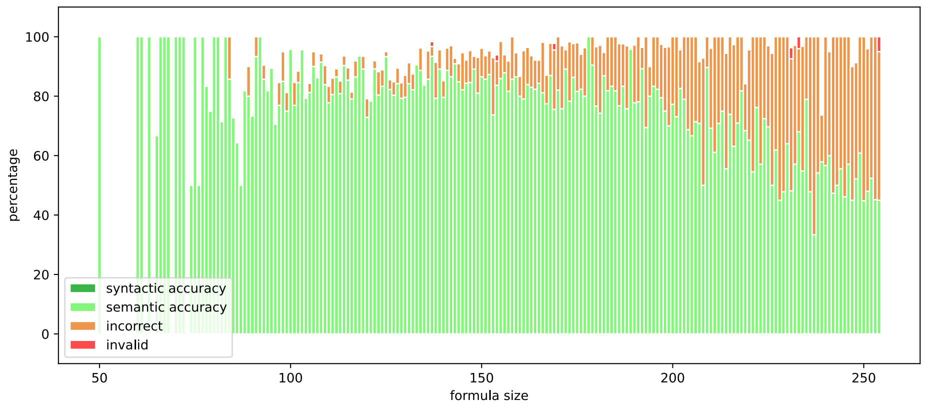


Figure 6.11: Predictions of our best performing model, trained on *LTLPattern126*, tested on 5704 specification patterns (*LTLUnsolved254*) for which spot timed out ($> 60s$). Semantic accuracy is displayed in green; incorrect traces in orange; syntactically invalid traces in red.

PSPACE), this experiment shows that the predictions of a deep neural network can be a valuable extension to the verification tool box.

Table 6.1 shows the effect of the most significant parameters on the performance of Transformers. The performance largely benefits from an increased number of layers, with 8 yielding the best results. Increasing the number further, even with much more training time, did not result in better or even led to worse results. A slightly less important role plays the number of heads and the dimension of the intermediate fully-connected feed-forward networks (FC). While a certain FC size is important, increasing it alone will not improve results. Changing the number of heads alone has also almost no impact on performance. Increasing both simultaneously, however, will result in a small gain. This seems reasonable, since more heads can provide more distinct information to the subsequent processing by the fully-connected feed-forward network. Increasing the embeddings size from 128 to 256 very slightly improves the syntactic accuracy. But likewise it also degrades the semantic accuracy, so we therefore stuck with the former setting.

6.4.2 Generalization Properties

To prove that the generalization to the semantics is independent of the data generation, we also trained a model on a dataset of randomly generated formulas. The *unshaded* part of Figure 6.12 displays the performance of our best model on the *LTLRandom*35 dataset. The Transformers were solely trained on formulas of size less

Embedding size	Layers	Heads	FC size	Batch Size	Train Steps	Syn. Acc.	Sem. Acc.
128	3	4	512	512	45K	78.0%	97.1%
128	5	2	512	512	45K	80.4%	97.4%
128	5	4	256	512	45K	81.0%	97.4%
128	5	4	512	512	45K	82.0%	97.9%
128	5	4	1024	512	45K	80.3%	97.3%
128	5	6	1024	512	45K	81.8%	97.7%
128	5	8	512	512	45K	82.0%	97.8%
128	5	8	1024	512	45K	82.5%	97.9%
128	5	8	1500	512	45K	82.6%	97.8%
128	5	12	1024	512	45K	81.9%	97.5%
128	8	4	512	512	45K	83.2%	98.3%
128	8	8	1024	768	50K	83.8%	**98.5%**
128	10	4	512	512	75K	82.9%	97.6%
256	5	4	512	512	45K	82.3%	97.9%

Table 6.1: Syntactic accuracy and semantic accuracy of different Transformers, tested on *LTLRandom*35: Layers refer to the size of the encoder and decoder stacks; Heads refer to the number of attention heads; FC size refers to the size of the fully-connected neural networks inside the encoder and decoders.

or equal to 35. We observe that in this range the exact syntactic accuracy decreases when the formulas grow in size. The semantic accuracy, however, stays, again, high. The model achieves a syntactic accuracy of 83.8% and a semantic accuracy of 98.5% on *LTLRandom*35, i.e., in 14.7% of the cases, the Transformer deviates from our automaton-based data generator. In Figure 6.13 we show the evolution of both the syntactic accuracy and the semantic accuracy during the training process. Note the significant difference right from the beginning. This demonstrates the importance of a suitable performance measure when evaluating machine learning algorithms on logical reasoning tasks.

To show that the generalization to larger formulas is independent from the data generation method, we also tested how well the Transformer generalizes to randomly generated LTL formulas of a size it has never seen before. We used our model trained on *LTLRandom*35 and observed the performance on *LTLRandom*50. The model preserves the semantic generalization, displayed in the *shaded* part of Figure 6.12. It outputs exact syntactic matches in 67.6% of the cases and achieves a semantic accuracy of 92.2%. For the generalization to larger formulas we utilized a positional encoding based on the tree representation of the formula [192]. The basic idea is to encode the path through the syntax tree for each character. Since LTL has only unary and binary operations, this is encoded by appending either $[1,0]$, representing the left child or $[0,1]$, representing the right child, in front of the encoding. When using the standard positional encoding instead, the accuracy drops, as expected, significantly. A visualization of this experiments can be found in Figure 6.14.

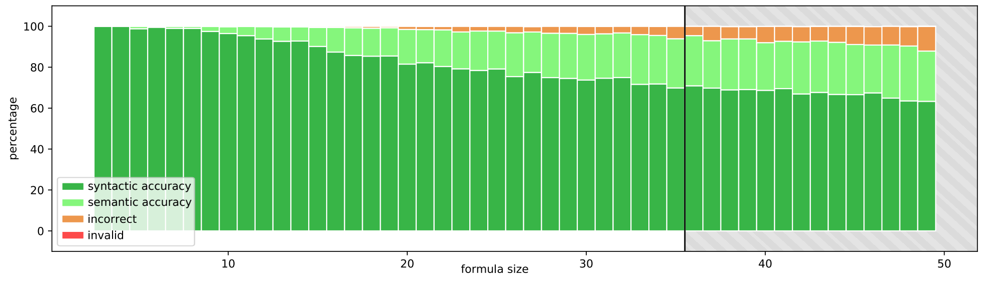


Figure 6.12: Syntactic and semantic accuracy of our best performing model (only trained on *LTLRandom35*) on *LTLRandom50*. Dark green is syntactically correct; light green is semantically correct, orange is incorrect.

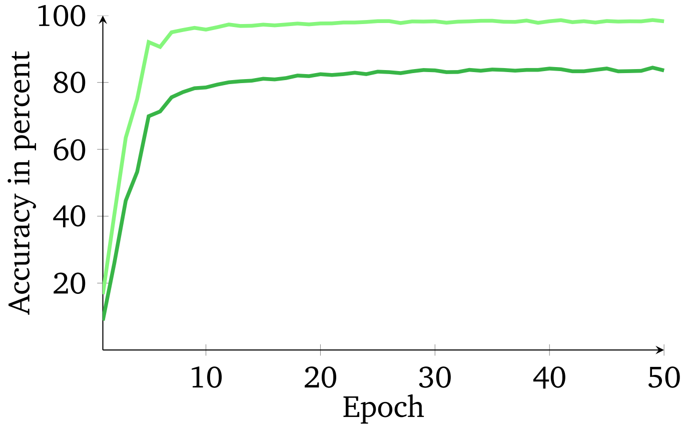


Figure 6.13: Syntactic accuracy (dark green) and semantic accuracy (light green) of our best performing model, evaluated on a subset of 5K samples of *LTLRandom*35 per epoch.

In a further experiment, we tested the out-of-distribution (OOD) generalization of the Transformer on the trace generation task. The Results of this experiment are displayed in Table 6.2. We generated a new dataset *LTLRandom*126 to match the formula sizes and the vocabulary of *LTLPattern*126. A model trained on *LTLRandom*126 achieves a semantic accuracy of 24.7% (and a syntactic accuracy of only 1.0%) when tested on *LTLPattern*126. Vice versa, a model trained on *LTLPattern*126 achieves a semantic accuracy of 38.5% (and a semantic accuracy of only 0.5%) when tested on *LTLRandom*126. Testing the models OOD increases the gap between syntactic and semantic correctness dramatically. This underlines that the models learned the nature of the LTL semantics rather than the generator process. Note that the two distributions are very different.

Following these observations, we also tested the performance of our models on

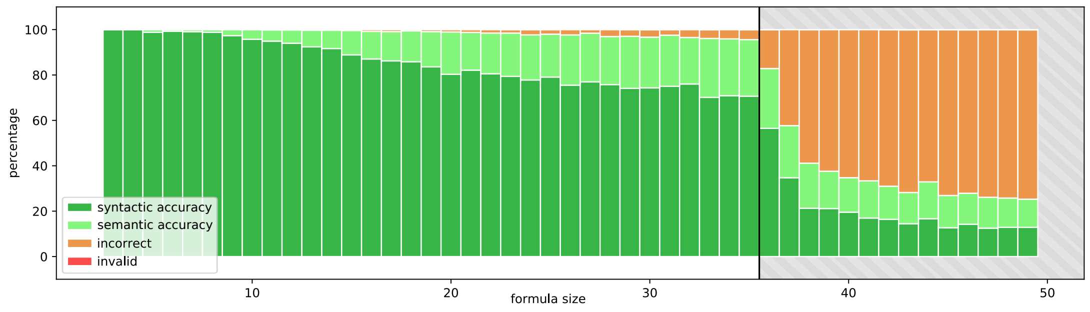


Figure 6.14: Performance of our best model (only trained on *LTLRandom*35) on *LTLRandom*50 with a standard positional encoding.

other patterns from the literature. We observe a higher semantic accuracy for our model trained on random formulas and a higher gap between semantic and syntactic accuracy for our model trained on pattern formulas (see Table 6.2).

In a last experiment on LTL, we tested the performance of our models on hand-crafted formulas.

The LTL formula $(b \, \mathrm{U} \, a) \wedge (a \, \mathrm{U} \, \neg a)$ states that b has to hold along the trace until a holds and a has to hold until a does not hold anymore. The automaton-based generator suggests the trace $(\neg a \wedge b) \; a \; (\mathit{true})^{\omega}$, i.e., to first satisfy the second until by immediately disallowing a. The satisfaction of the first until is then postponed to the second position of the trace, which forces b to hold on the first position. The Transformer, however, chooses the following more general trace $a \; (\neg a) \; (\mathit{true})^{\omega}$, by satisfying the until operators in order (see Figure 6.15, right).

We especially observed that formulas with multiple until statements that describe overlapping intervals were the most challenging. This is no surprise as these formulas are the source of PSPACE-hardness of LTL.

Patterns	Number of Patterns	Trained on	Syn. Acc.	Sem. Acc.
dac [53]	55	*LTLRandom*126	49.1%	81.8%
eh [61]	11	*LTLRandom*126	81.8%	90.9%
hkrss [110]	49	*LTLRandom*126	71.4%	83.7%
p [162]	20	*LTLRandom*126	65.0%	90.0%
eh [61]	11	*LTLPattern*126	0.0%	36.4%
hkrss [110]	49	*LTLPattern*126	14.3%	49.0%
p [162]	20	*LTLPattern*126	10.0%	60.0%

Table 6.2: Syntactic and semantic accuracy for OOD tests on different pattern formula data sets.

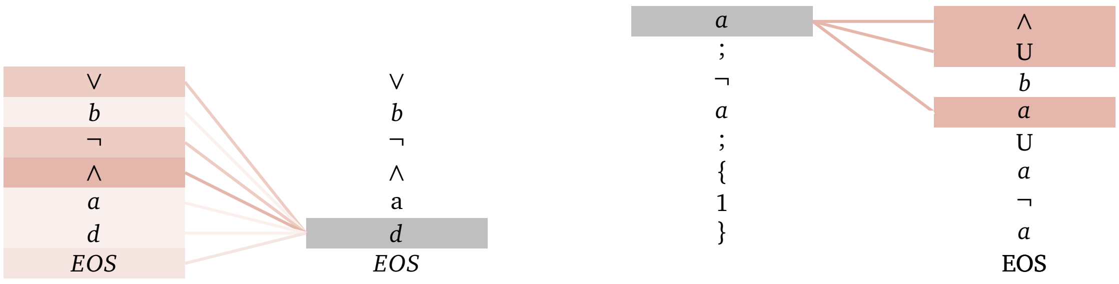


Figure 6.15: Encoder self-attention values of one attention head for the example propositional formula $b \vee \neg(a \wedge d)$ in dataset *PropRandom*35 (left). Encoder-decoder attention of the example LTL formula $(b \,\mathrm{U}\, a) \wedge (a \,\mathrm{U} \neg a)$ in dataset *LTLRandom*35 (right).

$a \,\mathrm{U}\, b \wedge a \,\mathrm{U} \neg b$	$(a \wedge \neg b)\ (b)\ (true)^{\omega}$
`&UabUa!b`	`&a!b;b;{1}`

While the above formula can be solved by most models, when scaling this formula to four overlapping until intervals, all of our models fail: For example, a model trained on *LTLRandom*35 predicted the trace $(a \wedge b \wedge c)\ (a \wedge \neg b \wedge \neg c)\ (b \wedge c)\ (true)^{\omega}$, which does not satisfy the LTL formula.

$(a \,\mathrm{U}\, b \wedge c) \wedge (a \,\mathrm{U} \neg b \wedge c) \wedge (a \,\mathrm{U}\, b \wedge \neg c) \wedge (a \,\mathrm{U} \neg b \wedge \neg c)$	$(a \wedge b \wedge c)\ (a \wedge \neg b \wedge \neg c)\ (b \wedge c)\ (true)^{\omega}$
`&&&Ua&bcUa&!bcUa&b!cUa&!b!c`	`&&abc;&&a!b!c;&bc;1`

6.4.3 Predicting Assignments for Propositional Logic

To show that the generalization to the semantic is not a specific property of LTL, we trained a Transformer to solve the assignment generation problem for propositional logic, which is a substantially different logical problem.

As a baseline for our generalization experiments on propositional logic, we trained and tested a model with the following hyperparameter on *PropRandom*35:

Embedding size	Layers	Heads	FC size	Batch Size	Train Steps	Syn. Acc.	Sem. Acc.
enc:128, dec:64	6	6	512	1024	50K	58.1%	**96.5%**

We observe a striking 38.4% gap between predictions that were syntactical matches of our DPLL-based generator and correct predictions of the Transformer. Only 3.5% of the time, the Transformer outputs an incorrect assignment. Note that we allow the derived operators $\oplus$ and $\leftrightarrow$ in these experiments, which succinctly represent complicated logical constructs.

The formula $b \vee \neg(a \wedge d)$ occurs in our dataset *PropRandom*35 and its corresponding assignment is $\{(a, 0)\}$. The Transformer, however, outputs d0, i.e., it goes with the assignment of setting d to *false*, which is also a correct solution. A visualization of this example can be found in Figure 6.15. When the formulas get larger, the solutions where the Transformer differs from the DPLL algorithm accumulate. Consider, for example, the formula $\neg b \vee (e \leftrightarrow b \vee c \vee \neg d) \vee (c \wedge (b \oplus (a \oplus \neg d)) \oplus (\neg c \leftrightarrow d) \wedge (a \leftrightarrow (b \oplus (b \oplus e))))$, which is also in the dataset *PropRandom*35. The generator suggests the assignment $\{(a,1),(c,1),(d,0)\}$. The Transformer, however, outputs e0, i.e., the singleton assignment of setting e to *false*, which turns out to be a (very small) solution as well.

We achieved stable training in this experiment by setting the decoder embedding size to either 64 or even 32. Keeping the decoder embedding size at 128 led to very unstable training.

We also tested whether the generalization to the semantics is preserved when the Transformer encounters propositional formulas of a larger size than it ever saw during training. We, again, utilized the tree positional encoding. When challenged with formulas of size 35 to 50, our best performing model trained on *PropRandom*35 achieves a syntactic accuracy of 35.8% and a semantic accuracy of 86.1%. In comparison, without the tree positional encoding, the Transformer achieves a syntactic match of only 29.0% and an overall accuracy of only 75.7%. Note that both positional encodings work equally well when not considering larger formulas.

In a last experiment, we tested how the Transformer performs on more challenging propositional formulas in CNF. We thus trained a model on *PropCNF*250, where it achieved a semantic accuracy of 65.1% and a syntactic accuracy of 56.6%. We observe a slightly lower gap compared to our LTL experiments. The Transformer, however, still deviates even on such formulas from the generator.

6.5 Summary

We trained a Transformer to predict solutions to linear-time temporal logical (LTL) formulas. We observed that our trained models evince powerful generalization properties, namely, the generalization to the semantics of the logic, and the generalization to larger formulas than seen during training. We showed that these generalizations do *not* depend on the underlying logical problem nor on the data generator. Regarding the performance of the trained models, we observed that they can compete with classical algorithms for generating solutions to LTL formulas. We built a test set that contained only formulas that were generated out of practical verification patterns, on which even our data generator timed out. Our best performing model, although it was trained on much smaller formulas, predicts correct traces 83% of the time.

Chapter 7

Neural Circuit Synthesis from Specification Patterns

The circuit synthesis problem for linear-time temporal logic (LTL) is the problem to construct a circuit implementation automatically from the LTL specification. Efficient synthesis tools for LTL would simplify the hardware design process significantly. A hardware designer could focus on specifying *what* the circuit is supposed to compute, instead of implementing *how* the computation is done. LTL synthesis procedures, however, have to invoke involved reasoning engines, which often turn out to be infeasible when facing real-world problem instances. Much research has been conducted to push this form of hardware construction process closer to practice (see, for example, the successful synthesis of the AMBA protocol [20]). The high computational complexity of the general problem (2-EXPTIME-complete [166]), however, is so far a barrier that seems insurmountable with classical, e.g., automaton-based, approaches. Recent successful applications of machine learning for logical tasks, such as SAT solving [188, 189], higher-order theorem proving [159, 13], and the LTL trace generation problem (see Chapter 6) encourage new approaches to the LTL synthesis problem using machine learning. Similar to the success of machine learning for program synthesis, e.g., [160, 100, 176], machine learning approaches might open a lot of possibilities in hardware synthesis. For example, secondary design goals, which cannot be easily formalized, might be incorporated into the process using natural language. Applying machine learning to the area of hardware synthesis, however, suffers from a severe lack of sufficient amounts of training data.

In this chapter, we build on the observation that deep neural networks generalize to the semantics of temporal logics. We consider a method to generate large amounts of additional training data, i.e., pairs of specifications and circuits implementing them. We show that hierarchical Transformers [141] can be trained on the circuit synthesis problem using the generated data and that the models can solve a significant portion of problems from the annual synthesis competition (SYNTCOMP) [116].

(assumptions)
$\square(\diamond(\neg(i_0)))$
$\bigcirc(\square((\neg(o_2))\lor(((\neg(i_4))\land(\neg(i_1)))$
$\mathrm{U}((\neg(i_4))\land(i_1)))))$
$\rightarrow$
(guarantees:)
$\square((i_0)\rightarrow(\diamond((\neg(i_0))\lor(o_4))))$
$\square((i_2)\rightarrow(\diamond(o_0)))$
$\square((i_1)\rightarrow(\diamond(o_0)))$
$(\square(\diamond(o_4)))\rightarrow(\square((\diamond(i_4))\land(\diamond(i_1))$
$\square((i_4)\rightarrow(\diamond(o_3)))$
$\bigcirc(\square((\neg(o_4))\lor(\neg(o_2))))$
$\square((o_1)\rightarrow(\bigcirc((i_1)\mathcal{R}(((i_1)$
$\rightarrow(o_2))\land((\neg(i_1))\rightarrow(o_0))))))$
$\square((\bigcirc(o_3))\rightarrow(i_3))$

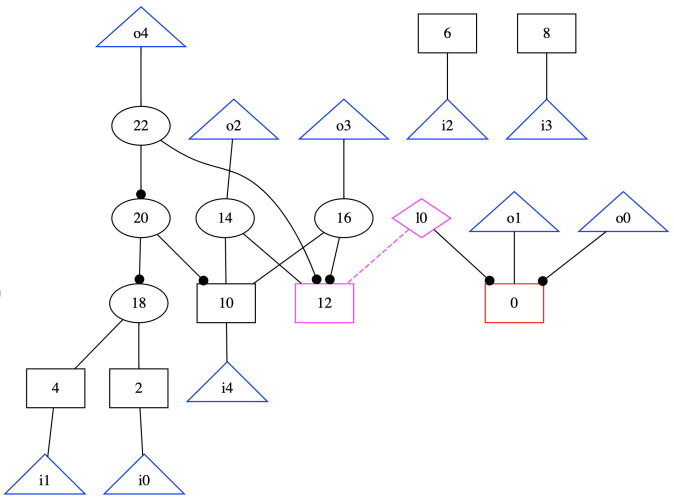


Figure 7.1: A specification in our test set, consisting of 2 assumption patterns and 8 guarantee patterns (left). A circuit, predicted by a hierarchical Transformer, satisfying the specification (right).

In practice, logical hardware specifications follow specific design patterns [53]. To cope with the data scarcity of this problem, we propose a method to use specification patterns, from which data for successful training can be derived. For example, a common LTL synthesis pattern looks as follows: $\square(r \rightarrow \diamond g)$. The formula describes a *response* property, stating that at every point in time ($\square$), a request r must be eventually ($\diamond$) followed by a grant g. We obtain these patterns from SYNTCOMP. We mined 2099 specification patterns from 346 benchmarks. By combining these patterns (see Section 7.1.2 for more details), we obtained 250000 specifications and used classical synthesis tools [67, 152] to compute circuits satisfying the specifications. Figure 7.1 shows an example held-out specification constructed in this fashion and a circuit predicted by one of our models (details on the data representation can be found in Section 7.1). When checking, the predicted circuit indeed satisfies the specification.

To train a machine learning model on the LTL synthesis task, we represent the decomposed specifications and circuits as sequences and use hierarchical Transformers [141]. We show that a hierarchical Transformer can successfully be trained on the LTL synthesis task using our data generation method. We show that many of the model's predictions that differ from the circuits in our dataset satisfy the specifications when verifying the predictions[1], i.e., the model constructs a different, yet cor-

[1]Verifying the solutions, i.e., model-checking, is an easier problem than synthesis (PSPACE [196] vs 2-EXPTIME [166]) and can typically be done in a fraction of the time needed to synthesize the circuits classically.

rect solution. When using a beam search, models achieve an accuracy of up to 78.7% on our synthetic test data and up to 67.6% on the original formulas from SYNTCOMP. The Transformer can even solve out-of-distribution formulas, taken from a recent case study [5], i.e., formulas that were not used for the specification pattern mining. Furthermore, the models can solve generated test instances on which classical LTL synthesis tools timed out. In practice, it is essential to handle both realizable (i.e., when a hardware implementation exists) and unrealizable (i.e., when no hardware implementation exists) specifications. We demonstrate that our approach achieves similar results on both realizable and unrealizable specifications.

Results in this chapter are based on "Neural Circuit Synthesis from Specification Patterns" [182], which was joint work with Frederik Schmitt, Markus N. Rabe, and Bernd Finkbeiner. The chapter is structured as follows. The data representation and generation process is described in Section 7.1. The experimental setup and the experimental evaluation are presented in Section 7.2 and Section 7.3, respectively. We provide a summary in Section 7.4.

7.1 Datasets

In the following, we first explain the circuit representation we use in this chapter. We then describe our dataset, which is mined from specification patterns from the LTL track of SYNTCOMP 2020 [118].

7.1.1 And-Inverter Graphs

The AIGER format [18] describes circuits as and-inverter graphs. It is widely used, especially for benchmarks and competitions in reactive synthesis. We base the following explanation on the explanation in the AIGER format report [18]. First, the header of an AIGER file defines the following:

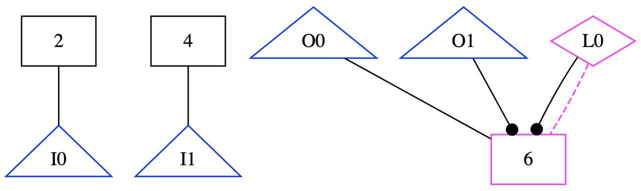
- M: the maximum variable index,
- I: the number of inputs,
- L: the number of latches,
- O: the number of outputs, and
- A: the number of AND gates.

Variables and literals are used for defining these. Literals are constants or signed variables represented by unsigned integers. To obtain a literal from a variable, the literal must be divided by 2. The literal modulo 2 defines if it corresponds to the negated or unnegated variable, respectively. The constant FALSE is represented by 0 and the constraint TRUE by 1. After the header, the next I lines define the inputs, which are defined as unnegated literals. The next L lines define the latches, which are two literals separated by a space character. The first literal denotes the current state of the latch and the second is the next state of the latch. The next O lines define the outputs, which can be arbitrary literals. The last A lines define the AND gates, which are three literals separated by a space character. The first integer must be even and represents the literal of the left-hand side (LHS). The two other integers represent the literals of the right-hand side (RHS) of the AND gate. After the necessary definitions, an optional symbol table and comments may follow. Symbols can be attached to inputs, latches, and outputs.

In the following, we repeat our small example from previous chapters. The inputs r_0, r_1 and outputs g_0, g_1 represent requests and grants of an arbiter. The following LTL formula specifies an arbiter:

$$\begin{aligned} &\square\neg(g_0 \wedge g_1) \\ \wedge\ &\square(r_0 \rightarrow \diamond g_0) \\ \wedge\ &\square(r_1 \rightarrow \diamond g_1)\ . \end{aligned}$$

The following AIGER representation (right) is a correct prediction for this specification of our hierarchical Transformer model. The visual representation of the circuit is given on the left.



aag 3 2 1 2 0	*header*
2	r_0
4	r_1
6 7	*latch*
6	g_0
7	g_1

The triangles represent inputs and outputs, the rectangles represent variables, the diamond-shaped variables represent latches and the black dots represent inverter (NOT-gates). The circuit implementation ignores the inputs $I0$ and $I1$, which represent both requests r_0 and r_1 (except for unnecessarily assigning them to variables 2 and 4). The circuit implementation satisfies the arbiter specification above by alternating indefinitely between both outputs $O0$ and $O1$, which represent both grants g_0 and g_1, independently of the given inputs. This is, in fact, the smallest solution satisfying the simple arbiter specification above. The hierarchical Transformer also predicts correct circuit implementations for more involved specifications where the circuit has to react to inputs (see, for example, Section 7.3 for an arbiter that prioritizes a certain request).

```
{
  "semantics": "mealy",
  "inputs": [
        "r_m",
        "r_0"
  ],
  "outputs": [
        "g_m",
        "g_0"
  ],
  "assumptions": [
        "(G (F (! (r_m))))"
  ],
  "guarantees": [
        "(true)",
        "(G ((! (g_m)) || (! (g_0))))",
        "(G ((r_0) -> (F (g_0))))",
        "(G ((r_m) -> (X ((! (g_0)) U (g_m)))))"
  ]
}
```

Figure 7.2: Specification of a prioritized arbiter in BoSy input format that is part of the 2020 SYNTCOMP benchmarks [118].

7.1.2 Data Generation

In the following section, we describe our data generation method. The basis of our data set are benchmarks from the LTL track of the annual reactive synthesis competition (SYNTCOMP 2020 [118]). We collected 346 benchmarks in the Temporal Logic Synthesis Format (TLSF) [117]. We translated the TLSF specifications to the BoSy input format [67] with SyFCo [117]. A specification in the BoSy input format consists of a list of assumptions and a list of guarantees. An assumption poses a restriction on the environment, and a guarantee defines how the system reacts to inputs. From these specification patterns, we generate larger specifications by conjoining assumption patterns to a specification φ_A and by conjoining guarantee patterns to a specification φ_G. The implication $\varphi_A \rightarrow \varphi_G$ forms the final specification of the circuit. An example of the format for a prioritized arbiter specification is shown in Figure 7.2.

We mined assumptions and guarantees of 346 SYNTCOMP benchmarks. We filtered LTL formulas with more than five inputs, more than five outputs, or an abstract syntax tree with size greater than 25 out. As a result, we constructed 157 assumption patterns and 1942 guarantee patterns. In a final step, we renamed inputs and outputs with a uniform random choice from input and output atomic propositions. The table below, shows three random examples of assumption patterns and three random examples of guarantee patterns.

assumption patterns	guarantee patterns
$\square(i_0 \wedge \bigcirc(\neg o_0 \wedge \neg o_1) \rightarrow \bigcirc i_0)$	$(o_2 \, \mathrm{U} \, i_3) \vee \square o_2$
$\square \Diamond i_0$	$\square(i_0 \rightarrow \bigcirc(o_3 \vee i_3 \vee \bigcirc(o_3 \vee i_3 \vee \bigcirc(o_3 \vee i_3))))$
$\square(\neg i_0 \vee o_3 \vee o_2 \vee o_1 \vee o_0 \vee \bigcirc i_0)$	$\square(\neg o_2 \vee \neg o_4)$

We constructed our data set out of these specification patterns as follows. We generated pairs of LTL specifications in the BoSy format and circuits implementing them. The specifications are constructed by alternating between sampling guarantees until the specification becomes unrealizable and sampling assumptions until the specification becomes realizable. Finding a suitable assumption is tried 5 times. Furthermore, we implemented stopping criteria that limit the maximal number of guarantees to 10, the maximal number of assumptions to 3, and the runtime for the synthesis tool to 120 seconds. If the resulting specification is unrealizable we also consider its realizable predecessor for our dataset. Apart from that intermediate specifications are discarded. To synthesize specifications, we use the LTL synthesis tool Strix [152]. Systems are represented in the AIGER format. For unrealizable specifications we provide an AIGER circuit representing the winning strategy for the environment, i.e., a counter strategy showing that the specification is unrealizable. To make sure that the data fits into our models, we additionally filter the circuits as follows. We filter circuits exceeding a maximum variable index of 50 and circuits with k AND gates if the number of circuits in the dataset with k AND gates exceeds 20% of the dataset size.

```
aag 11 5 1 5
5
2
4
6
8
10
12 1
1
0
14
16
22
14 12 10
16 13 10
18 4 2
20 19 11
22 21 13
```

Figure 7.3: AIGER representation of the circuit in Figure 7.1.

With our data generation method, we constructed a data set with 250000 specification-circuit-pairs. The data set is split into 80% training samples, 10% validation samples, and 10% test samples. The benefit of this data generation method is that we can generate large amounts of specifications from practical, thus meaningful, patterns. The number of underlying patterns can be comparatively small, as long as they contain the main ideas of the domain at hand (such as the SYNTCOMP benchmarks for reactive synthesis). To, furthermore, be able to predict unrealizability, and, thus, counter strategies, we also included unrealizable specifications. They are constructed through the first stopping criteria. For balance, we generated the data

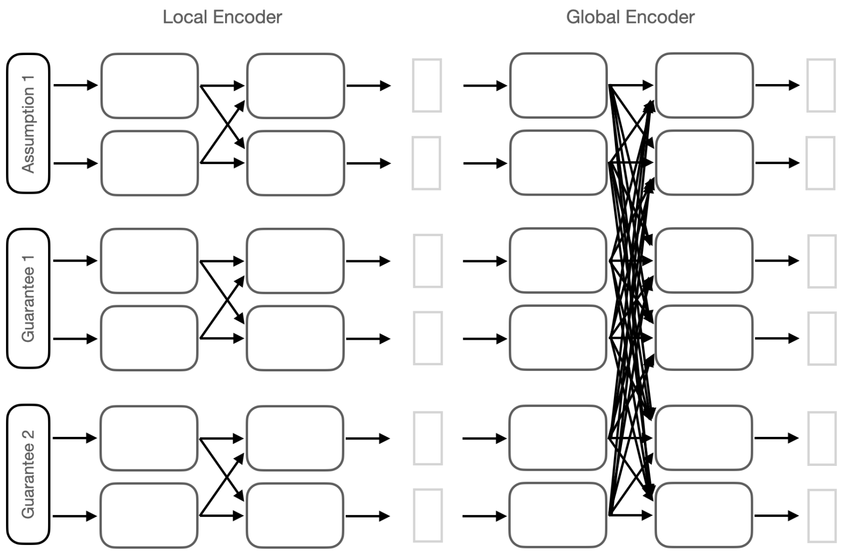


Figure 7.4: A hierarchical Transformer (HAT) [141] first encodes assumption and guarantee patterns in isolation, before encoding them globally.

set such that half of the dataset consists of unrealizable specifications. An example instance of the data set is shown in Figure 7.1 and Figure 7.3. Figure 7.1 depicts the specification and the visualization of the circuit and Figure 7.3 shows the AIGER representation of this circuit.

7.2 Experimental Setup

We implemented a hierarchical Transformer (HAT) [141] and augment it with a tree-positional encoding [192].[2] In contrast to a baseline Transformer, the encoder has two types of layers, local and global layers.

The local layers encode individual assumptions and guarantees, and only the global layers can combine the representations of tokens across all assumptions and all guarantees. With this hierarchical encoding, we gain approximately 10% of accuracy across all models compared to using a standard Transformer (see Figure 7.5). Figure 7.4 sketches the use of local and global layers in the encoder for our setting.

We trained hierarchical Transformers with model dimension 256. The dimension of the feed-forward networks was set to 1024. The encoder employs 4 local layers followed by 4 global layers, and the decoder employs 8 (unmodified) layers. All our attention layers use 4 attention heads. We trained with a batch size of 256 for 30000

[2]The code, our data sets, and data generators are part of the Python library ML2 (`https://github.com/reactive-systems/ml2`).

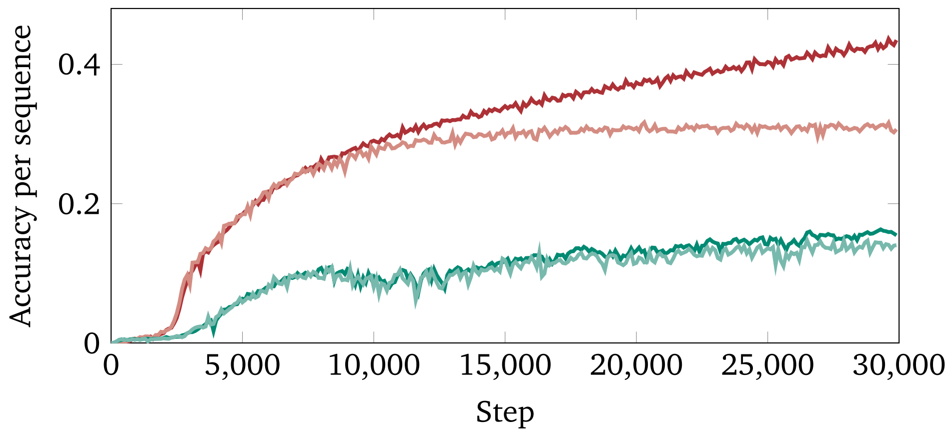


Figure 7.5: Accuracy per sequence over the training course shown for the training split (red) and validation split (light red) when training the hierarchical Transformer and for the training split (green) and the validation split (light green) when training the standard Transformer.

steps ($\sim$ 39 epochs). We saved the model with the best accuracy per sequence on the validation data. We trained on an NVIDIA DGX A100 system for around 10 hours.

7.2.1 Training Details

The Transformer architecture (see Section 6.1) is a sequence-to-sequence model trained to predict a sequence of output tokens provided a sequence of input tokens. Similarly, we provide multiple sequences of input tokens to an hierarchical Transformer. Assumptions and guarantees are LTL formulas and can thus be directly represented as sequences of tokens with each atomic proposition, Boolean operator, temporal operator, and Boolean constant being a separate token. We omit parentheses because we add, as for the trace generation problem in the previous chapter, a tree-positional encoding [192] that identifies each token with its position in the abstract syntax tree of the LTL formula. To distinguish assumptions from guarantees in the global step we prepend assumptions with a special assumption token. Circuits are in AIGER format that we represent as a sequence of tokens by representing each digit with a corresponding token and replacing each newline character with a special new line token. Since all circuits in our dataset have the same inputs and outputs we can omit the header and the symbol table when tokenizing an AIGER circuit. Additionally, we include a special realizability token at the beginning of the sequence indicating whether a specification is realizable.

We used the Adam optimizer [125] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$ and a learning rate schedule as proposed in [214]. The learning rate is increased linearly for the *warmup steps*. The learning rate then decreases proportionally to the inverse square root of the step.

7.2.2 Performance Measures

Similar to the experiments on the trace generation problem of LTL in Chapter 6, we have to measure the performance semantically. For a given realizable LTL specification, there are infinitely many circuits implementing them. Thus, more than one prediciton of the Transformer might be correct. We use nuXmv [34] to check the predictions of our models. As in Chapter 6, we, thus, distinguish between the syntactic and semantic accuracy. The percentage of the Transformer's predictions that satisfy the specification is called semantic accuracy. In the next section, we show that even for the LTL synthesis problem, which is "harder" compared to the trace generation problem, the Transformer model still comes up with own solutions.

7.3 Experiments

In this section, we report on a variety of experiments that analyze the performance of hierarchical Transformers on the circuit synthesis task and their generalization behavior. In the following, we will first analyze the overall performance of the models and see that they often construct different solutions, yet correct ones, than the classical tool we generated the training data with. For this, we consider four different test sets and group results on the size of the predicted circuits. Secondly, we compare the training with our data mining method against the ground truth, i.e., against a training of a hierarchical Transformer on the raw SYNTCOMP benchmarks. Thirdly, we compare the models performance on realizable and unrealizable specifications. Lastly, we will take a deeper look into one of the specifications, which, compared to the example in Section 7.1, is an arbiter that prioritizes a certain request.

Dataset	Beam Size 1	Beam Size 4	Beam Size 8	Beam Size 16
`Testset`	53.1 (31.0)	69.5 (39.9)	74.3 (42.6)	78.7 (45.8)
`SYNTCOMP`	51.7	60.7	63.4	67.6
`Timeouts`	12.6	20.8	26.1	31.1
`SmartHome`	19.0	33.3	33.3	47.6

Table 7.1: Accuracy reported on test data, SYNTCOMP benchmarks, timeouts, and smart home benchmarks for different beam sizes. For the test data we show the syntactic accuracy in parenthesis.

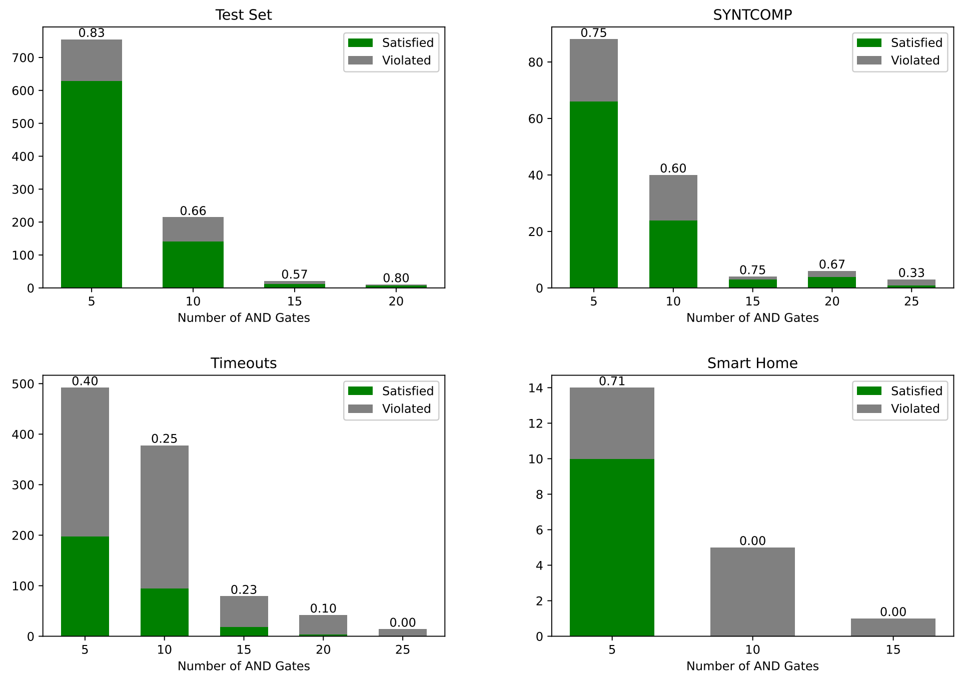


Figure 7.6: Accuracy with respect to the size of the synthesized circuits measured by the number of AND gates for test set (top, left), SYNTCOMP (top, right), timeouts (bottom, left), and smart home benchmarks (bottom, right). Number of AND gates are binned into intervals of size 5.

Overall results. We tested our models on four different datasets. A `Testset` consisting of held-out instances generated by our data mining method, the `SYNTCOMP` set, consisting of the synthesis competition benchmarks, a set `Timeouts` that consists of generated specifications on which Strix [152], the classical synthesis tool that we used for generating the circuits, timed out ($< 120s$), and an out-of-distribution (OOD) benchmark set `SmartHome` consisting of specifications for smart homes [5]. We consistently observed in all experiments that the beam search significantly increases the accuracy. When analyzing the results we found that the beam search often yields several correct circuits. For a beam size of 16 and the `Testset`, on average 5 of the 16 AIGER circuits satisfy the specification.

In our `Testset` (see Table 7.1), we observe in many cases that the circuit prediction of our model is different from the circuit the tool would synthesize. Since we already showed this gap between syntactic and semantic accuracy in the previous chapter, we concentrate on the semantic accuracy, i.e., the total accuracy. We found no significant decrease or increase in average circuit size. In total, the model was able to solve 78.7% of the held-out generated test instances with a beam size of 16.

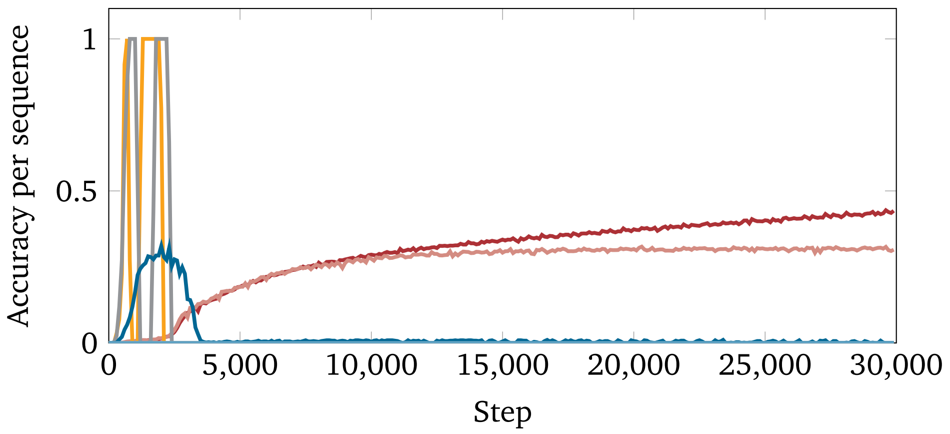


Figure 7.7: Accuracy per sequence over the training course shown for the training split (red) and validation split (light red) of the synthetic dataset and the training split and the validation split (not visible, $\sim$ 0% accuracy per sequence) of the raw SYNTCOMP dataset: orange, gray, and blue for a batch size of 256, 64, and 8, respectively.

While the training data is based on specification patterns extracted from SYNTCOMP benchmarks it is unlikely that our data generation process reassembles SYNTCOMP benchmarks. This allows to evaluate the model on them. After filtering out benchmarks with more than 5 inputs/outputs, more than 12 properties, and properties of size greater than 25, the model achieved an accuracy of 67.6% for the resulting 145 benchmarks using a beam size of 16.

For a timed out specification it is not known whether it is realizable or unrealizable. The model achieves an accuracy of 31.1% for beam size 16. This demonstrates that our approach can yield performance gains in practice. To highlight the capabilities of our model we display one of the largest correctly predicted circuit for a timed out specification in Figure 7.8.

We constructed the `SmartHome` set with the same restriction as for the `SYNTCOMP` set. The recently published benchmark set consists of specifications for synthesizing smart home applications [5]. The hierarchical Transformer is able to solve 47.6% of the provided instances. When compared to the full benchmark (i.e., without the size restrictions), the model solved 11.1% of the formulas. Note that this benchmark set was not used to mine specifications from and the benchmarks include instances with larger assumptions and guarantees than seen during training.

We also analyzed the performance of the model depending on the size of the predicted circuit. Results are shown in Figure 7.6. As expected, for larger circuit implementations, the model accuracy drops. The size distribution of the training data resembles the size distribution of the test set (top left in Figure 7.6 and Figure 7.10).

```
aag 21 5 2 5
14
2
4
6
8
10
12 17
14 43
17
0
0
19
0
16 15 12
18 15 13
20 13 8
22 15 9
24 23 21
26 25 7
28 6 3
30 28 20
32 31 27
34 33 5
36 4 3
38 36 7
40 38 20
42 41 35
```

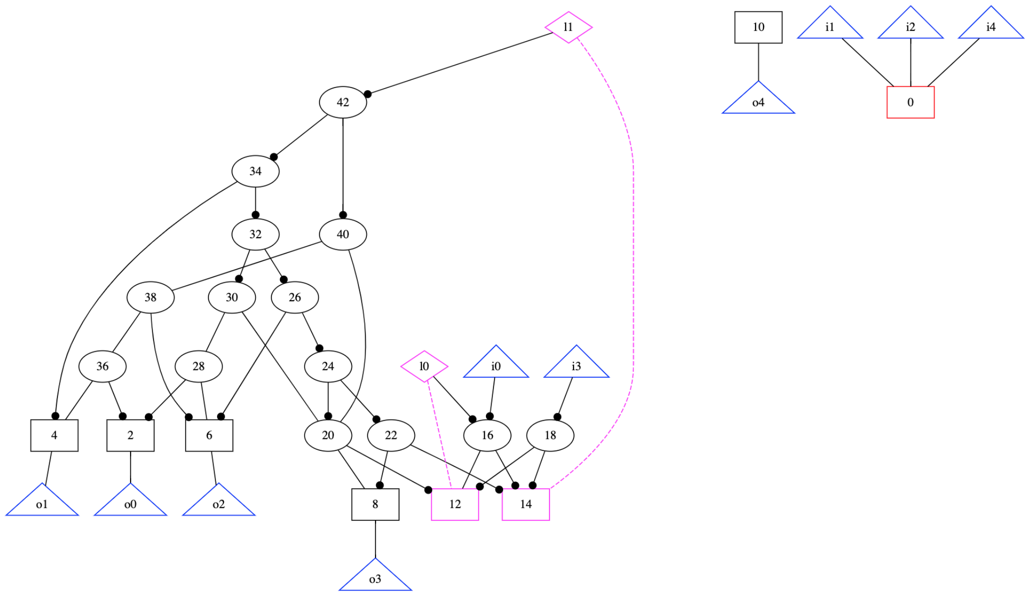


Figure 7.8: One of the largest circuit that satisfies a specification on which the classical tool times out.

(assumptions)
$(\Box(\Diamond\neg(r_m)))$
$\rightarrow$
(guarantees:)
$(\Box((\neg(g_m))\vee(\neg(g_0))))$
$(\Box((r_0)\rightarrow(\Diamond(g_0))))$
$(\Box((r_m)\rightarrow(\bigcirc((\neg(g_0))\,\mathrm{U}\,(g_m)))))$

```
aag 7 5 1 5 1
2
4
6
8
10
12 4
14
0
13
14
0
14 12 5
```

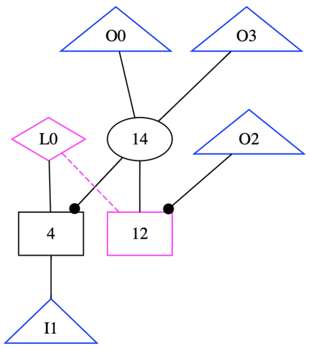


Figure 7.9: The specification (left), the predicted AIGER circuit (middle) and the visualization of the circuit (right) for a prioritizing arbiter.

Note that the model has seen a significantly lower percentage of large circuits during training.

Training on raw SYNTCOMP benchmarks. We also provide a baseline experiment. Figure 7.7 shows that our data generation methods enables a stable training while (not surprisingly) training only on the raw SYNTCOMP benchmarks fails.

Unrealizabile Specifications. The training data contains both realizable and unrealizable specifications. In Table 7.2 we analyze the accuracy for realizable and unrealizable specifications separately on our test data. While the syntactic accuracy is higher for realizable specifications, in terms of the semantic accuracy the model solves unrealizable specifications slightly more accurately. Further, we found for a beam size of 1 that the Transformer predicts the correct realizability token for 91.4% of the specifications from the test data.

	Beam Size 1	Beam Size 4	Beam Size 8	Beam Size 16
Realizable	50.8 (39.0)	64.3 (48.0)	67.5 (50.0)	70.7 (52.6)
Unrealizable	55.4 (23.0)	74.6 (31.9)	81.0 (35.2)	86.7 (39.0)

Table 7.2: Accuracy on `Testset` reported separately for realizable and unrealizable specifications. For different beam sizes we report the semantic accuracy and the syntactic accuracy in parenthesis.

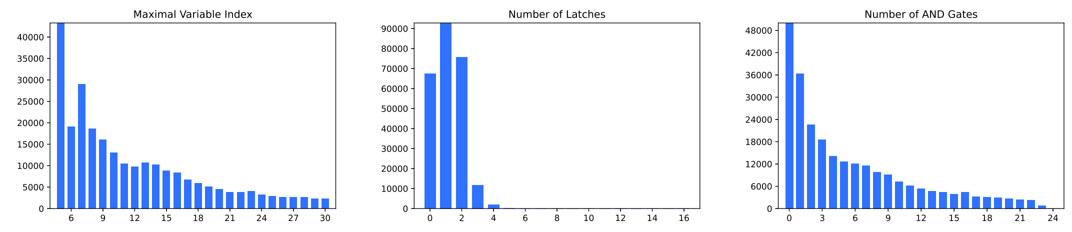


Figure 7.10: Distribution of maximal variable index, number of latches, and number of AND gates in the dataset.

Prioritizing arbiter. Building on the example of Section 7.1, we show that the model can handle more interesting, real-world specifications. Figure 7.9 shows the specification, AIGER file, and the circuit visualization of an arbiter that prioritizes one of the requests whenever access is requested by both processes at the same time. This means that the circuit implementation can no longer ignore the input, by simply alternating between the different grants, as for the example in Section 7.1.

Conclusion to Experimental Results. Our experimental results show that the hierarchical Transformer can be more than a valuable addition to the LTL synthesis toolbox. The results do not only encourage the implementation of Transformer models in formal method tools, but also provide a method for constructing hardware directly out of specifications. For example, one could use a Transformer model as an end-to-end heuristic by simply verifying the predictions. Experimental results suggest that the Transformer can be especially useful for predicting unrealizability.

7.4 Summary

We proposed a method to address the lack of data for training a neural network on the task of synthesizing circuits out of LTL specifications. We mine specification patterns from the annual reactive synthesis competition and generate new formulas by combining multiple specification patterns. We showed that this dataset can be used to successfully train hierarchical Transformers on the LTL synthesis problem for specifications composed of specification patterns. We also showed that the models generalize to unseen specifications, including specifications that are both realizable and unrealizable and specifications that cannot be solved by a classical synthesis tool within a time limit of 120 seconds. Furthermore, we performed an out-of-distribution test on a recently added benchmark set on synthesis problems for smart homes. The data generation method proposed in this chapter already enables the training of machine learning models for replacing classical heuristics in LTL synthesis tools.

Chapter 8

Conclusion

In this thesis, we studied logical and deep learning methods for the temporal reasoning about reactive systems. We first studied temporal hyperproperties expressed in a hyperlogic. We considered purely logical methods, i.e., satisfiability, realizability, and enforcement of temporal hyperproperties. We provided solutions to these problems for highly expressive hyperlogics. Secondly, towards coping with the high computational cost of these problems, we trained a neural network on the trace and circuit generation task for the baseline temporal logic (LTL). We showed that a neural network could solve challenging formal methods problems, such as constructing a circuit out of an LTL specification, end-to-end. In the following sections, we summarize the results of this thesis before concluding.

8.1 Part 1: Logical Methods for Temporal Reasoning

In the first part of this thesis, we identified decidable fragments of the satisfiability and realizability problem of expressive hyperlogics. On the linear-time spectrum, we studied HyperQPTL, a temporal logic for ω-regular hyperproperties. On the branching-time spectrum, we studied HyperCTL*, a temporal logic for branching-time hyperproperties. Both logics are more expressive than the baseline hyperlogic HyperLTL. As an application of the satisfiability and realizability problem of hyperproperties, we studied the enforcement problem of temporal hyperproperties. We provided algorithms for enforcing HyperLTL specifications in a reactive black-box system. Experimental results showed that enforcing hyperproperties at runtime is feasible, and the bottleneck is the a prio solving of the parity game.

With the results of Part I of this thesis, we determined the borders for which logical methods for temporal hyperproperties remain decidable in theory and feasible in practice. The results provide a foundational understanding of the hierarchy of hyperlogics. The presented algorithms and decision procedures lay the foundation for practical tools extending the toolkit for the automated analysis of hyperproperties.

8.2 Part 2: Deep Learning Methods for Temporal Reasoning

In the second part of this thesis, we trained a Transformer, a deep neural network architecture, on the trace generation and realizability problem of linear-time temporal logic (LTL). With our data generation method, which constructs specifications and their solutions (traces or circuits) from specification patterns, we showed that the Transformer generalizes to the semantics of the temporal logic. Our experiments showed that these generalizations are also preserved when considering larger specifications than seen during training. The machine learning models were also capable of providing solutions to instances where classical solvers timed out. Overall, we provided the first end-to-end supervised learning approach for the trace generation and the circuit synthesis problem of LTL.

The results of Part II of this thesis suggest that deep learning can already augment combinatorial approaches in automatic verification and the broader formal methods community. With these results, we can, for example, derive novel algorithms for LTL synthesis that first query a Transformer for circuit predictions. One of the main benefits of the approach to learn from LTL specifications is that the predictions can be checked comparatively easy. This enables the development of hybrid algorithms: Classical methods can serve as a fallback, check partial solutions guiding the Transformer, or verify predictions, e.g., in a reinforcement loop. This approach, however, also constitutes new challenges, such as the acquisition of large amounts of data.

8.3 Concluding Remarks

Reactive systems are increasingly linked through the Internet and, additionally, constantly grow in size. The potential that arises from studying temporal logical reasoning under these circumstances is immense: Treating privacy and security requirements equally important as classical functional requirements will be essential in the future. Furthermore, deep learning techniques have proven themselves to provide significant efficiency gains for many applications. Deep learning for temporal logics, thus, holds the promise to empower researchers in the automated reasoning and formal methods communities to make bigger jumps in developing new automated verification methods.

In addition to the contributions that deep learning brings to formal methods, deep learning techniques, and artificial intelligence (AI) in general, could also benefit from formal methods techniques. Formal specification languages serve as a link between requirements in natural language and formal specifications, which can be utilized for automated reasoning. Understanding and reasoning on natural language are crucial towards artificial general intelligence (AGI). A formal specification language, such as a temporal logic, could also serve as a link to natural language in the context of AI research. The benefit of this intermediate step is the ability to verify predictions of the neural network automatically. Automatically verifiable predictions also significantly increase interpretability towards more explainable AI (XAI). Thus, further studies on machine learning techniques for formal (temporal) specification languages could be a worthwhile step towards these goals.

Bibliography

[1] 2019. A. M. Turing Award 1996 Amir Pnueli. Retrieved July 15, 2021 from https://amturing.acm.org/award_winners/pnueli_4725172.cfm

[2] 2019. *Meltdown and Spectre*. Retrieved July 15, 2021 from https://meltdownattack.com/

[3] 2021. *AbsInt Angewandte Informatik GmbH*. Retrieved July 15, 2021 from https://www.absint.com/

[4] 2021. *Gödel Prize 2000 Moshe Y. Vardi and Pierre Wolper*. Retrieved July 15, 2021 from https://eatcs.org/index.php/component/content/article/508

[5] 2021. *J.A.R.V.I.S. TSL/TLSF Benchmark Suite*. Retrieved July 15, 2021 from https://github.com/SYNTCOMP/benchmarks/tree/master/tlsf/tsl_smart_home_jarvis

[6] 2021. *The John Locke Lectures*. Retrieved July 15, 2021 from https://www.philosophy.ox.ac.uk/john-locke-lectures

[7] Shreya Agrawal and Borzoo Bonakdarpour. 2016. Runtime Verification of k-Safety Hyperproperties in HyperLTL. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 239–252. https://doi.org/10.1109/CSF.2016.24

[8] J. Alammar. 2018. *The Illustrated Transformer*. Retrieved July 15, 2021 from https://jalammar.github.io/illustrated-transformer/

[9] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJOFETxR-

[10] Gilles Audemard and Laurent Simon. 2018. On the Glucose SAT Solver. *Int. J. Artif. Intell. Tools* 27, 1 (2018), 1840001:1–1840001:25. `https://doi.org/10.1142/S0218213018400018`

[11] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. *CoRR* abs/1607.06450 (2016). arXiv:1607.06450 `http://arxiv.org/abs/1607.06450`

[12] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. 2018. Learning to Solve SMT Formulas. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 10338–10349. `https://proceedings.neurips.cc/paper/2018/hash/68331ff0427b551b68e911eebe35233b-Abstract.html`

[13] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 454–463. `http://proceedings.mlr.press/v97/bansal19a.html`

[14] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Math. Struct. Comput. Sci.* 21, 6 (2011), 1207–1252. `https://doi.org/10.1017/S0960129511000193`

[15] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4 (2011), 14:1–14:64. `https://doi.org/10.1145/2000799.2000800`

[16] Catriel Beeri. 1980. On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases. *ACM Trans. Database Syst.* 5, 3 (1980), 241–259. `https://doi.org/10.1145/320613.320614`

[17] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 60–70. `https://doi.org/10.1145/3180155.3180219`

[18] Armin Biere. 2007. The AIGER and-inverter graph (AIG) format version 20071012. *FMV Reports Series, Institute for Formal Models and Verifica-*

tion, Johannes Kepler University, Altenbergerstr 69 (2007), 4040. https://doi.org/10.35011/fmvtr.2007-1

[19] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016. Hammering towards QED. *J. Formaliz. Reason.* 9, 1 (2016), 101–148. https://doi.org/10.6092/issn.1972-5787/4593

[20] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. 2007. Automatic Hardware Synthesis From Specifications: A Case Study. In *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*. EDA Consortium, San Jose, CA, USA, 1188–1193. https://dl.acm.org/citation.cfm?id=1266622

[21] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. 2015. Shield Synthesis: - Runtime Enforcement for Reactive Systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*. Springer, 533–548. https://doi.org/10.1007/978-3-662-46681-0_51

[22] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. 2012. Acacia+, a Tool for LTL Synthesis. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 652–657. https://doi.org/10.1007/978-3-642-31424-7_45

[23] Borzoo Bonakdarpour and Bernd Finkbeiner. 2018. The Complexity of Monitoring Hyperproperties. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 162–174. https://doi.org/10.1109/CSF.2018.00019

[24] Borzoo Bonakdarpour and Bernd Finkbeiner. 2019. Program Repair for Hyperproperties. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11781)*. Springer, 423–441. https://doi.org/10.1007/978-3-030-31784-3_25

[25] Borzoo Bonakdarpour, César Sánchez, and Gerardo Schneider. 2018. Monitoring Hyperproperties by Combining Static Analysis and Runtime Verification. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol,*

Cyprus, November 5-9, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11245). Springer, 8–27. https://doi.org/10.1007/978-3-030-03421-4_2

[26] Laura Bozzelli, Bastien Maubert, and Sophie Pinchinat. 2015. Unifying Hyper and Epistemic Temporal Logics. In *Proceedings of FoSSaCS (LNCS, Vol. 9034)*. Springer, 167–182. https://doi.org/10.1007/978-3-662-46678-0_11

[27] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. 2017. Massive Exploration of Neural Machine Translation Architectures. *CoRR* abs/1703.03906 (2017). arXiv:1703.03906 http://arxiv.org/abs/1703.03906

[28] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[29] J Richard Buchi. 1963. Weak second-order arithmetic and finite automata. *Journal of Symbolic Logic* 28, 1 (1963). https://doi.org/10.1002/malq.19600060105

[30] J Richard Büchi. 1990. On a decision method in restricted second order arithmetic. In *The collected works of J. Richard Büchi*. Springer, 425–435. https://doi.org/10.1007/978-1-4613-8928-6_23

[31] J Richard Buchi and Lawrence H Landweber. 1969. Solving Sequential Conditions by Finite-State Strategies. *Trans. Amer. Math. Soc.* 138 (1969), 295–311. https://doi.org/10.2307/1994916

[32] Paul A. Cairns. 2004. Informalising Formal Mathematics: Searching the Mizar Library with Latent Semantics. In *Mathematical Knowledge Management, Third International Conference, MKM 2004, Bialowieza, Poland, September 19-21,*

2004, Proceedings (Lecture Notes in Computer Science, Vol. 3119). Springer, 58–72. https://doi.org/10.1007/978-3-540-27818-4_5

[33] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. 2017. Deciding parity games in quasipolynomial time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*. ACM, 252–263. https://doi.org/10.1145/3055399.3055409

[34] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*. Springer, 334–342. https://doi.org/10.1007/978-3-319-08867-9_22

[35] David Chaum. 1985. Security Without Identification: Transaction Systems to Make Big Brother Obsolete. *Commun. ACM* 28, 10 (1985), 1030–1044. https://doi.org/10.1145/4372.4373

[36] Alonzo Church. 1963. Application of recursive arithmetic to the problem of circuit synthesis. (1963). https://doi.org/10.2307/2271310

[37] Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981 (Lecture Notes in Computer Science, Vol. 131)*. Springer, 52–71. https://doi.org/10.1007/BFb0025774

[38] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (1986), 244–263. https://doi.org/10.1145/5397.5399

[39] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2001. *Model checking*. MIT Press. http://books.google.de/books?id=Nmc4wEaLXFEC

[40] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8414)*, Martín Abadi and

Steve Kremer (Eds.). Springer, 265–284. https://doi.org/10.1007/978-3-642-54792-8_15

[41] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (2010), 1157–1210. https://doi.org/10.3233/JCS-2009-0393

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

[43] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Yannick Schillo. 2021. Runtime Enforcement of Hyperproperties. In *19th International Symposium on Automated Technology for Verification and Analysis (ATVA)*.

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

[45] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2018. Model Checking Boot Code from AWS Data Centers. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*. Springer, 467–486. https://doi.org/10.1007/978-3-319-96142-2_28

[46] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[47] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint*

European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963). Springer, 337–340. `https://doi.org/10.1007/978-3-540-78800-3_24`

[48] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 109–124. `https://doi.org/10.1109/SP.2010.15`

[49] Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah. 2020. Probabilistic Hyperproperties of Markov Decision Processes. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12302)*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer, 484–500. `https://doi.org/10.1007/978-3-030-59152-6_27`

[50] Alexandre Donzé, Thomas Ferrère, and Oded Maler. 2013. Efficient Robust Monitoring for STL. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 264–279. `https://doi.org/10.1007/978-3-642-39799-8_19`

[51] Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. 2019. Compositional Falsification of Cyber-Physical Systems with Machine Learning Components. *J. Autom. Reason.* 63, 4 (2019), 1031–1053. `https://doi.org/10.1007/s10817-018-09509-5`

[52] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 - A Framework for LTL and ω -Automata Manipulation. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9938)*. 122–129. `https://doi.org/10.1007/978-3-319-46520-3_8`

[53] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1998. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*. ACM, 7–15. `https://doi.org/10.1145/298595.298598`

[54] Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10482)*. Springer, 269–286. `https://doi.org/10.1007/978-3-319-68167-2_19`

[55] Rüdiger Ehlers and Bernd Finkbeiner. 2011. Monitoring Realizability. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7186)*. Springer, 427–441. `https://doi.org/10.1007/978-3-642-29860-8_34`

[56] Rüdiger Ehlers and Bernd Finkbeiner. 2017. Symmetric Synthesis. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India (LIPIcs, Vol. 93)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:13. `https://doi.org/10.4230/LIPIcs.FSTTCS.2017.26`

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

[58] E. Allen Emerson and Charanjit S. Jutla. 1991. Tree Automata, Mu-Calculus and Determinacy (Extended Abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE Computer Society, 368–377. `https://doi.org/10.1109/SFCS.1991.185392`

[59] Úlfar Erlingsson and Fred B. Schneider. 1999. SASI enforcement of security policies: a retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms, Caledon Hills, ON, Canada, September 22-24, 1999*. ACM, 87–95. `https://doi.org/10.1145/335169.335201`

[60] Javier Esparza, Jan Kretínský, Jean-François Raskin, and Salomon Sickert. 2017. From LTL and Limit-Deterministic Büchi Automata to Deterministic Parity Automata. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205)*. 426–442. `https://doi.org/10.1007/978-3-662-54577-5_25`

[61] Kousha Etessami and Gerard J. Holzmann. 2000. Optimizing Büchi Automata. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1877)*. Springer, 153–167. https://doi.org/10.1007/3-540-44618-4_13

[62] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. 2018. Can Neural Networks Understand Logical Entailment?. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SkZxCk-0Z

[63] Yliès Falcone. 2010. You Should Better Enforce Than Verify. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6418)*. Springer, 89–105. https://doi.org/10.1007/978-3-642-16612-9_9

[64] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2012. What can you verify and enforce at runtime? *Int. J. Softw. Tools Technol. Transf.* 14, 3 (2012), 349–382. https://doi.org/10.1007/s10009-011-0196-8

[65] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. 2011. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods Syst. Des.* 38, 3 (2011), 223–262. https://doi.org/10.1007/s10703-011-0114-4

[66] Yliès Falcone and Srinivas Pinisetty. 2019. On the Runtime Enforcement of Timed Properties. In *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11757)*, Bernd Finkbeiner and Leonardo Mariani (Eds.). Springer, 48–69. https://doi.org/10.1007/978-3-030-32079-9_4

[67] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. 2017. BoSy: An Experimentation Framework for Bounded Synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*. Springer, 325–332. https://doi.org/10.1007/978-3-319-63390-9_17

[68] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1ersoRqtm

[69] Bernd Finkbeiner. 2017. Temporal Hyperproperties. *Bull. EATCS* 123 (2017). http://eatcs.org/beatcs/index.php/beatcs/article/view/514

[70] Bernd Finkbeiner and Christopher Hahn. 2016. Deciding Hyperproperties. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada (LIPIcs, Vol. 59)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:14. https://doi.org/10.4230/LIPIcs.CONCUR.2016.13

[71] Bernd Finkbeiner, Christopher Hahn, and Tobias Hans. 2018. MGHyper: Checking Satisfiability of HyperLTL Formulas Beyond the $\exists^*\forall^*$ Fragment. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11138)*. Springer, 521–527. https://doi.org/10.1007/978-3-030-01090-4_31

[72] Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Leander Tentrup. 2020. Realizing ω-regular Hyperproperties. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225)*. Springer, 40–63. https://doi.org/10.1007/978-3-030-53291-8_4

[73] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. 2018. Synthesizing Reactive Systems from Hyperproperties. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 289–306. https://doi.org/10.1007/978-3-319-96145-3_16

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

[75] Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. 2017. EAHyper: Satisfiability, Implication, and Equivalence Checking of Hyperproperties. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*. Springer, 564–570. https://doi.org/10.1007/978-3-319-63390-9_29

[76] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2018. RVHyper: A Runtime Verification Tool for Temporal Hyperproperties. In

Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10806). Springer, 194–200. https://doi.org/10.1007/978-3-319-89963-3_11

[77] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2019. Monitoring hyperproperties. *Formal Methods Syst. Des.* 54, 3 (2019), 336–363. https://doi.org/10.1007/s10703-019-00334-z

[78] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2020. Efficient monitoring of hyperproperties using prefix trees. *Int. J. Softw. Tools Technol. Transf.* 22, 6 (2020), 729–740. https://doi.org/10.1007/s10009-020-00552-5

[79] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. 2018. Model Checking Quantitative Hyperproperties. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 144–163. https://doi.org/10.1007/978-3-319-96145-3_8

[80] Bernd Finkbeiner, Christian Müller, Helmut Seidl, and Eugen Zalinescu. 2017. Verifying Security Policies in Multi-agent Workflows with Loops. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 633–645. https://doi.org/10.1145/3133956.3134080

[81] Bernd Finkbeiner and Markus N. Rabe. 2014. The linear-hyper-branching spectrum of temporal logics. *it Inf. Technol.* 56, 6 (2014), 273–279. http://www.degruyter.com/view/j/itit.2014.56.issue-6/itit-2014-1067/itit-2014-1067.xml

[82] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL ˆ*. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*. Springer, 30–48. https://doi.org/10.1007/978-3-319-21690-4_3

[83] Bernd Finkbeiner and Sven Schewe. 2005. Uniform Distributed Synthesis. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29*

June 2005, Chicago, IL, USA, Proceedings. IEEE Computer Society, 321–330. `https://doi.org/10.1109/LICS.2005.53`

[84] Bernd Finkbeiner and Sven Schewe. 2013. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 519–539. `https://doi.org/10.1007/s10009-012-0228-z`

[85] Bernd Finkbeiner and Henny Sipma. 2004. Checking Finite Traces Using Alternating Automata. *Formal Methods Syst. Des.* 24, 2 (2004), 101–127. `https://doi.org/10.1023/B:FORM.0000017718.28096.48`

[86] Bernd Finkbeiner and Leander Tentrup. 2014. Detecting Unrealizable Specifications of Distributed Systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*. Springer, 78–92. `https://doi.org/10.1007/978-3-642-54862-8_6`

[87] Bernd Finkbeiner and Leander Tentrup. 2015. Detecting Unrealizability of Distributed Fault-tolerant Systems. *Log. Methods Comput. Sci.* 11, 3 (2015). `https://doi.org/10.2168/LMCS-11(3:12)2015`

[88] Bernd Finkbeiner and Martin Zimmermann. 2017. The First-Order Logic of Hyperproperties. In *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany (LIPIcs, Vol. 66)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:14. `https://doi.org/10.4230/LIPIcs.STACS.2017.30`

[89] Marie Fortin, Louwe B. Kuijer, Patrick Totzke, and Martin Zimmermann. 2021. HyperLTL Satisfiability is Σ_1^1-complete, HyperCTL* Satisfiability is Σ_1^2-complete. *CoRR* abs/2105.04176 (2021). arXiv:2105.04176 `https://arxiv.org/abs/2105.04176`

[90] World Economic Forum. 2020. Future Series: Cybersecurity, emerging technology and systemic risk. (2020). `http://www3.weforum.org/docs/WEF_Future_Series_Cybersecurity_emerging_technology_and_systemic_risk_2020.pdf`

[91] Oliver Friedmann and Martin Lange. 2009. The PGSolver collection of parity game solvers. *University of Munich* (2009), 4–6. `https://www.win.tue.nl/~timw/downloads/amc2014/pgsolver.pdf`

[92] Paul Gastin and Denis Oddoux. 2001. Fast LTL to Büchi Automata Translation. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2102)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, 53–65. `https://doi.org/10.1007/3-540-44585-4_6`

[93] Paul Gastin and Denis Oddoux. 2021. *LTL2BA*. Retrieved July 15, 2021 from `http://www.lsv.fr/~gastin/ltl2ba/`

[94] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. 2021. TacticToe: Learning to Prove with Tactics. *J. Autom. Reason.* 65, 2 (2021), 257–286. `https://doi.org/10.1007/s10817-020-09580-x`

[95] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 3–18. `https://doi.org/10.1109/SP.2018.00058`

[96] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1263–1272. `http://proceedings.mlr.press/v70/gilmer17a.html`

[97] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 11–20. `https://doi.org/10.1109/SP.1982.10014`

[98] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press. `http://www.deeplearningbook.org/`

[99] Lars Grunske. 2008. Specification patterns for probabilistic quality properties. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 31–40. `https://doi.org/10.1145/1368088.1368094`

[100] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin G. Zorn. 2015. Inductive program-

ming meets the real world. *Commun. ACM* 58, 11 (2015), 90–99. https://doi.org/10.1145/2736282

[101] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, 1345–1351. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[102] Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. 2021. Teaching Temporal Logics to Neural Networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=dOcQK-f4byz

[103] Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2019. Constraint-Based Monitoring of Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11428)*. Springer, 115–131. https://doi.org/10.1007/978-3-030-17465-1_7

[104] Joseph Y. Halpern and Moshe Y. Vardi. 1989. The Complexity of Reasoning about Knowledge and Time. I. Lower Bounds. *J. Comput. Syst. Sci.* 38, 1 (1989), 195–237. https://doi.org/10.1016/0022-0000(89)90039-1

[105] Tobias Hans. 2021. *Algorithms for Deciding HyperCTL**. Master's thesis. Saarland University.

[106] David Harel and Amir Pnueli. 1984. On the Development of Reactive Systems. In *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984 (NATO ASI Series, Vol. 13)*. Springer, 477–498. https://doi.org/10.1007/978-3-642-82453-1_17

[107] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. 1026–1034. https://doi.org/10.1109/ICCV.2015.123

[108] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. `https://openreview.net/forum?id=B1lnbRNtwr`

[109] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780. `https://doi.org/10.1162/neco.1997.9.8.1735`

[110] Jan Holeček, Tomáš Kratochvíla, Vojtěch Řehák, David Šafránek, Pavel Šimeček, et al. 2004. Verification results in Liberouter project. `https://www.fi.muni.cz/~xrehak/publications/verificationresults.pdf`

[111] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. 2021. Bounded Model Checking for Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12651)*. Springer, 94–112. `https://doi.org/10.1007/978-3-030-72016-2_6`

[112] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A Learning Environment for Theorem Proving. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. `https://openreview.net/forum?id=r1xwKoR9Y7`

[113] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*. Springer, 3–29. `https://doi.org/10.1007/978-3-319-63387-9_1`

[114] IEEE-Commission et al. 2005. IEEE standard for property specification language (PSL). *IEEE Std 1850-2005* (2005). `https://standards.ieee.org/standard/1850-2010.html`

[115] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. DeepMath - Deep Sequence Models for Premise Selection. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*.

2235–2243. https://proceedings.neurips.cc/paper/2016/hash/f197002b9a0853eca5e046d9ca4663d5-Abstract.html

[116] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. 2017. The first reactive synthesis competition (SYNTCOMP 2014). *Int. J. Softw. Tools Technol. Transf.* 19, 3 (2017), 367–390. https://doi.org/10.1007/s10009-016-0416-3

[117] Swen Jacobs, Felix Klein, and Sebastian Schirmer. 2016. A High-Level LTL Synthesis Format: TLSF v1.1. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016 (EPTCS, Vol. 229)*. 112–132. https://doi.org/10.4204/EPTCS.229.10

[118] Sven Jacobs and Guillermo A. Pérez. 2020. *The Reactive Synthesis Competition SYNTCOMP 2020 Results*. Retrieved July 15, 2021 from http://www.syntcomp.org/syntcomp-2020-results/

[119] Roope Kaivola. 1997. Using automata to characterise fixed point temporal logics. (1997). https://www.lfcs.inf.ed.ac.uk/reports/97/ECS-LFCS-97-356/

[120] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir A. Frolov, Erik Reeber, and Armaghan Naik. 2009. Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation. In *Intl. Conference on Computer Aided Verification*. Springer, 414–429. https://doi.org/10.1007/978-3-642-02658-4_32

[121] Cezary Kaliszyk, François Chollet, and Christian Szegedy. 2017. HolStep: A Machine Learning Dataset for Higher-order Logic Theorem Proving. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ryuxYmvel

[122] Cezary Kaliszyk and Josef Urban. 2014. Learning-Assisted Automated Reasoning with Flyspeck. *J. Autom. Reason.* 53, 2 (2014), 173–213. https://doi.org/10.1007/s10817-014-9303-3

[123] Johan Anthony Wilem Kamp. 1968. *Tense logic and the theory of linear order*. University of California, Los Angeles. https://www.proquest.com/dissertations-theses/tense-logic-theory-linear-order/docview/302320357/se-2?accountid=12586

[124] Yonit Kesten and Amir Pnueli. 1995. A Complete Proof Systems for QPTL. In *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26-29, 1995*. IEEE Computer Society, 2–12. https://doi.org/10.1109/LICS.1995.523239

[125] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[126] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1–19. https://doi.org/10.1109/SP.2019.00002

[127] Sascha Konrad and Betty H. C. Cheng. 2005. Real-time specification patterns. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. ACM, 372–381. https://doi.org/10.1145/1062455.1062526

[128] Dexter Kozen. 1983. Results on the Propositional mu-Calculus. *Theor. Comput. Sci.* 27 (1983), 333–354. https://doi.org/10.1016/0304-3975(82)90125-6

[129] Andreas Krebs, Arne Meier, Jonni Virtema, and Martin Zimmermann. 2018. Team Semantics for the Specification and Verification of Hyperproperties. In *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK (LIPIcs, Vol. 117)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:16. https://doi.org/10.4230/LIPIcs.MFCS.2018.10

[130] Lars Kuhtz and Bernd Finkbeiner. 2009. LTL Path Checking Is Efficiently Parallelizable. In *Automata, Languages and Programming, 36th Internatilonal Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 5556)*. Springer, 235–246. https://doi.org/10.1007/978-3-642-02930-1_20

[131] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. 2009. From liveness to promptness. *Formal Methods in System Design* 34, 2 (2009), 83–103. https://doi.org/10.1007/s10703-009-0067-z

[132] Orna Kupferman and Moshe Y. Vardi. 1999. Model Checking of Safety Properties. In *Computer Aided Verification, 11th International Conference, CAV '99,*

Trento, Italy, July 6-10, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1633), Nicolas Halbwachs and Doron A. Peled (Eds.). Springer, 172–183. https://doi.org/10.1007/3-540-48683-6_17

[133] Orna Kupfermant and Moshe Y Vardit. 2000. Synthesis with incomplete informatio. In *Advances in temporal logic*. Springer, 109–127. http://dx.doi.org/10.1007/978-94-015-9586-5_6

[134] Guillaume Lample and François Charton. 2020. Deep Learning For Symbolic Mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=S1eZYeHFDS

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

[136] Gil Lederman, Markus N. Rabe, Sanjit Seshia, and Edward A. Lee. 2020. Learning Heuristics for Quantified Boolean Formulas through Reinforcement Learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=BJluxREKDB

[137] Dennis Lee, Christian Szegedy, Markus N. Rabe, Sarah M. Loos, and Kshitij Bansal. 2020. Mathematical Reasoning in Latent Space. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=Ske31kBtPr

[138] Jianwen Li, Yinbo Yao, Geguang Pu, Lijun Zhang, and Jifeng He. 2014. Aalta: an LTL satisfiability checker over Infinite/Finite traces. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, 731–734. https://doi.org/10.1145/2635868.2661669

[139] Jianwen Li, Lijun Zhang, Geguang Pu, Moshe Y. Vardi, and Jifeng He. 2013. LTL Satisfiability Checking Revisited. In *2013 20th International Symposium on Temporal Representation and Reasoning, Pensacola, FL, USA, September 26-28, 2013*. IEEE Computer Society, 91–98. https://doi.org/10.1109/TIME.2013.19

[140] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. 2020. Modelling High-Level Mathematical Reasoning in Mechanised Declarative Proofs. *CoRR*

abs/2006.09265 (2020). arXiv:2006.09265 https://arxiv.org/abs/2006.09265

[141] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. 2021. IsarStep: a Benchmark for High-level Mathematical Reasoning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Pzj6fzU6wkj

[142] Xiao Li, Jessilyn Dunn, Denis Salins, Gao Zhou, Wenyu Zhou, Sophia Miryam Schüssler-Fiorenza Rose, Dalia Perelman, Elizabeth Colbert, Ryan Runge, Shannon Rego, et al. 2017. Digital Health: Tracking Physiomes and Activity Using Wearable Biosensors Reveals Useful Health-Related Information. *PLoS Biology* (2017). https://doi.org/10.1371/journal.pbio.2001402

[143] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJiHXGWAZ

[144] Jay Ligatti, Lujo Bauer, and David Walker. 2009. Run-Time Enforcement of Nonsafety Policies. *ACM Trans. Inf. Syst. Secur.* 12, 3 (2009), 19:1–19:41. https://doi.org/10.1145/1455526.1455532

[145] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[146] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. 2017. Deep Network Guided Proof Search. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017 (EPiC Series in Computing, Vol. 46)*. EasyChair, 85–105. https://easychair.org/publications/paper/ND13

[147] Qingzhou Luo and Grigore Rosu. 2013. EnforceMOP: a runtime property enforcement system for multithreaded programs. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. ACM, 156–166. https://doi.org/10.1145/2483760.2483766

[148] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3253)*. Springer, 152–166. https://doi.org/10.1007/978-3-540-30206-3_12

[149] Corto Mascle and Martin Zimmermann. 2020. The Keys to Decidable HyperLTL Satisfiability: Small Models or Very Simple Formulas. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain (LIPIcs, Vol. 152)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:16. https://doi.org/10.4230/LIPIcs.CSL.2020.29

[150] George H Mealy. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34, 5 (1955), 1045–1079. https://doi.org/10.1002/j.1538-7305.1955.tb03788.x

[151] Jia Meng and Lawrence C. Paulson. 2009. Lightweight relevance filtering for machine-generated resolution problems. *J. Appl. Log.* 7, 1 (2009), 41–57. https://doi.org/10.1016/j.jal.2007.07.004

[152] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. 2018. Strix: Explicit Reactive Synthesis Strikes Back!. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 578–586. https://doi.org/10.1007/978-3-319-96145-3_31

[153] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. http://arxiv.org/abs/1301.3781

[154] Edward F Moore. 1956. Gedanken-experiments on sequential machines. In *Automata Studies.(AM-34), Volume 34*. Princeton University Press, 129–154. https://doi.org/10.1515/9781400882618-006

[155] Matej Moravcík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael H. Bowling. 2017. DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker.

CoRR abs/1701.01724 (2017). arXiv:1701.01724 http://arxiv.org/abs/1701.01724

[156] Minh Ngo, Fabio Massacci, Dimiter Milushev, and Frank Piessens. 2015. Runtime Enforcement of Security Policies on Black Box Reactive Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 43–54. https://doi.org/10.1145/2676726.2676978

[157] Luan Viet Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V. Deshmukh, and Taylor T. Johnson. 2017. Hyperproperties of real-valued signals. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*. ACM, 104–113. https://doi.org/10.1145/3127041.3127058

[158] Gethin Norman and Vitaly Shmatikov. 2006. Analysis of probabilistic contract signing. *J. Comput. Secur.* 14, 6 (2006), 561–589. http://content.iospress.com/articles/journal-of-computer-security/jcs268

[159] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2020. Graph Representations for Higher-Order Logic and Theorem Proving. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2967–2974. https://aaai.org/ojs/index.php/AAAI/article/view/5689

[160] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJ0JwFcex

[161] Pawel Parys. 2019. Parity Games: Zielonka's Algorithm in Quasi-Polynomial Time. In *44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019, August 26-30, 2019, Aachen, Germany (LIPIcs, Vol. 138)*, Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:13. https://doi.org/10.4230/LIPIcs.MFCS.2019.10

[162] Radek Pelánek. 2007. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4595)*. Springer, 263–267. https://doi.org/10.1007/978-3-540-73370-6_17

[163] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015 (JMLR Workshop and Conference Proceedings, Vol. 37)*. JMLR.org, 1093–1102. http://proceedings.mlr.press/v37/piech15.html

[164] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of Reactive(1) Designs. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3855)*. Springer, 364–380. https://doi.org/10.1007/11609773_24

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

[166] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 179–190. https://doi.org/10.1145/75277.75293

[167] Amir Pnueli and Roni Rosner. 1990. Distributed Reactive Systems Are Hard to Synthesize. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*. IEEE Computer Society, 746–757. https://doi.org/10.1109/FSCS.1990.89597

[168] Stanislas Polu and Ilya Sutskever. 2020. Generative Language Modeling for Automated Theorem Proving. *CoRR* abs/2009.03393 (2020). arXiv:2009.03393 https://arxiv.org/abs/2009.03393

[169] Emil L Post. 1946. A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc.* 52, 4 (1946), 264–268. https://www.ams.org/journals/bull/1946-52-04/S0002-9904-1946-08555-9/S0002-9904-1946-08555-9.pdf

[170] A. N. PRIOR. 1955. *Time and Modality*. Greenwood Press. https://doi.org/10.2307/2216989

[171] Markus N. Rabe. 2016. *A Temporal Logic Approach to Information-Flow Control*. Ph.D. Dissertation. Saarland University.

[172] Markus Norman Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. 2021. Mathematical Reasoning via Self-supervised Skip-tree Training. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=YmqAnY0CMEy

[173] Markus N. Rabe and Christian Szegedy. 2021. Towards the Automatic Mathematician. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*. Springer, 25–37. https://doi.org/10.1007/978-3-030-79876-5_2

[174] Matthieu Renard, Yliès Falcone, Antoine Rollet, Thierry Jéron, and Hervé Marchand. 2019. Optimal enforcement of (timed) properties with uncontrollable events. *Math. Struct. Comput. Sci.* 29, 1 (2019), 169–214. https://doi.org/10.1017/S0960129517000123

[175] Kristin Y. Rozier and Moshe Y. Vardi. 2007. LTL Satisfiability Checking. In *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4595)*. Springer, 149–167. https://doi.org/10.1007/978-3-540-73370-6_11

[176] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html

[177] Shmuel Safra. 1988. On the Complexity of omega-Automata. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 319–327. https://doi.org/10.1109/SFCS.1988.21948

[178] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. 2019. Analysing Mathematical Reasoning Abilities of Neural Models. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA,*

USA, May 6-9, 2019. OpenReview.net. https://openreview.net/forum?id=H1gR5iR5FX

[179] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[180] Sven Schewe and Bernd Finkbeiner. 2006. Synthesis of Asynchronous Systems. In *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4407)*. Springer, 127–142. https://doi.org/10.1007/978-3-540-71410-1_10

[181] Imanol Schlag, Paul Smolensky, Roland Fernandez, Nebojsa Jojic, Jürgen Schmidhuber, and Jianfeng Gao. 2019. Enhancing the Transformer with Explicit Relational Encoding for Math Problem Solving. *CoRR* abs/1910.06611 (2019). arXiv:1910.06611 http://arxiv.org/abs/1910.06611

[182] Frederik Schmitt, Christopher Hahn, Markus Rabe, and Bernd Finkbeiner. 2021. Neural circuit synthesis from specification patterns. *Advances in Neural Information Processing Systems* 34 (2021).

[183] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50. https://doi.org/10.1145/353323.353382

[184] Stephan Schulz. 2013. System Description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8312)*. Springer, 735–743. https://doi.org/10.1007/978-3-642-45221-5_49

[185] Viktor Schuppan and Luthfi Darmawan. 2011. Evaluating LTL Satisfiability Solvers. In *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6996)*. Springer, 397–413. https://doi.org/10.1007/978-3-642-24372-1_28

[186] Stefan Schwendimann. 1998. A New One-Pass Tableau Calculus for PLTL. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '98, Oisterwijk, The Netherlands, May 5-8, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1397)*. Springer, 277–292. https://doi.org/10.1007/3-540-69778-0_28

[187] Dana Scott and Jacobus Willem de Bakker. 1969. A theory of programs. *Unpublished manuscript, IBM, Vienna* (1969).

[188] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11628)*. Springer, 336–353. https://doi.org/10.1007/978-3-030-24258-9_24

[189] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2019. Learning a SAT Solver from Single-Bit Supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=HJMC_iA5tm

[190] Sanjit A. Seshia, Ankush Desai, Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte, and Xiangyu Yue. 2018. Formal Specification for Deep Neural Networks. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11138)*. Springer, 20–34. https://doi.org/10.1007/978-3-030-01090-4_2

[191] Sanjit A. Seshia and Dorsa Sadigh. 2016. Towards Verified Artificial Intelligence. *CoRR* abs/1606.08514 (2016). arXiv:1606.08514 http://arxiv.org/abs/1606.08514

[192] Vighnesh Leonardo Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 12058–12068. https://proceedings.neurips.cc/paper/2019/hash/6e0917469214d8fbd8c517dcdc6b8dcf-Abstract.html

[193] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nat.* 550, 7676 (2017), 354–359. https://doi.org/10.1038/nature24270

[194] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.* 3, POPL (2019), 41:1–41:30. `https://doi.org/10.1145/3290354`

[195] Aravinda Prasad Sistla. 1983. *Theoretical issues in the design and verification of distributed systems*. Ph.D. Dissertation. `http://ra.adm.cs.cmu.edu/anon/usr0/ftp/usr/anon/home/anon/scan/CMU-CS-83-146.pdf`

[196] A. Prasad Sistla and Edmund M. Clarke. 1985. The Complexity of Propositional Linear Temporal Logics. *J. ACM* 32, 3 (1985), 733–749. `https://doi.org/10.1145/3828.3837`

[197] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. 1987. The Complementation Problem for Büchi Automata with Appplications to Temporal Logic. *Theor. Comput. Sci.* 49 (1987), 217–237. `https://doi.org/10.1016/0304-3975(87)90008-9`

[198] Sandro Stucki, César Sánchez, Gerardo Schneider, and Borzoo Bonakdarpour. 2019. Gray-Box Monitoring of Hyperproperties. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11800)*. Springer, 406–424. `https://doi.org/10.1007/978-3-030-30942-8_25`

[199] Christian Szegedy. 2020. A Promising Path Towards Autoformalization and General Artificial Intelligence. In *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12236)*. Springer, 3–20. `https://doi.org/10.1007/978-3-030-53518-6_1`

[200] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. `http://arxiv.org/abs/1312.6199`

[201] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. 1701–1708. `https://doi.org/10.1109/CVPR.2014.220`

[202] Leander Tentrup, Alexander Weinert, and Martin Zimmermann. 2016. Approximating Optimal Bounds in Prompt-LTL Realizability in Doubly-

exponential Time. In *Proceedings of GandALF (EPTCS, Vol. 226)*. 302–315. https://doi.org/10.4204/EPTCS.226.21

[203] Josef Urban. 2004. MPTP - Motivation, Implementation, First Experiments. *J. Autom. Reason.* 33, 3-4 (2004), 319–339. https://doi.org/10.1007/s10817-004-6245-1

[204] Josef Urban. 2007. MaLARea: a Metasystem for Automated Reasoning in Large Theories. In *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories, Bremen, Germany, 17th July 2007 (CEUR Workshop Proceedings, Vol. 257)*. CEUR-WS.org. http://ceur-ws.org/Vol-257/05_Urban.pdf

[205] Josef Urban and Jan Jakubuv. 2020. First Neural Conjecturing Datasets and Experiments. In *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12236)*. Springer, 315–323. https://doi.org/10.1007/978-3-030-53518-6_24

[206] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jirí Vyskocil. 2008. MaLARea SG1- Machine Learner for Automated Reasoning with Semantic Guidance. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5195)*. Springer, 441–456. https://doi.org/10.1007/978-3-540-71070-7_37

[207] Moshe Y. Vardi. 1995. Alternating Automata and Program Verification. In *Computer Science Today: Recent Trends and Developments*. Lecture Notes in Computer Science, Vol. 1000. Springer, 471–485. https://doi.org/10.1007/BFb0015261

[208] Moshe Y. Vardi. 2001. Branching vs. Linear Time: Final Showdown. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2031)*. Springer, 1–22. https://doi.org/10.1007/3-540-45319-9_1

[209] Moshe Y. Vardi. 2011. Branching vs. Linear Time: Semantical Perspective. In *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings (LIPIcs, Vol. 12)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3. https://doi.org/10.4230/LIPIcs.CSL.2011.3

[210] Moshe Y. Vardi and Larry J. Stockmeyer. 1985. Improved Upper and Lower Bounds for Modal Logics of Programs: Preliminary Report. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*. ACM, 240–251. https://doi.org/10.1145/22145.22173

[211] Moshe Y. Vardi and Pierre Wolper. 1986. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. IEEE Computer Society, 332–344.

[212] Moshe Y. Vardi and Pierre Wolper. 1994. Reasoning About Infinite Computations. *Inf. Comput.* 115, 1 (1994), 1–37. https://doi.org/10.1006/inco.1994.1092

[213] Ashish Vaswani, Samy Bengio, Eugene Brevdo, François Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Lukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. 2018. Tensor2Tensor for Neural Machine Translation. In *Proceedings of the 13th Conference of the Association for Machine Translation in the Americas, AMTA 2018, Boston, MA, USA, March 17-21, 2018 - Volume 1: Research Papers*. Association for Machine Translation in the Americas, 193–199. https://www.aclweb.org/anthology/W18-1819/

[214] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[215] Marcell Vazquez-Chanlatte. 2018. *mvcisback/py-aiger*. https://doi.org/10.5281/zenodo.1326224

[216] Jesse Vig. 2019. A Multiscale Visualization of Attention in the Transformer Model. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28 - August 2, 2019, Volume 3: System Demonstrations*. 37–42. https://www.aclweb.org/anthology/P19-3007/

[217] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embeddings for Program Repair. In *6th International Conference on Learning*

Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net. `https://openreview.net/forum?id=BJuWrGW0Z`

[218] Pierre Wolper. 1981. Temporal Logic Can Be More Expressive. In *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*. IEEE Computer Society, 340–348. `https://doi.org/10.1109/SFCS.1981.44`

[219] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. 2021. TacticZero: Learning to Prove Theorems from Scratch with Deep Reinforcement Learning. *CoRR* abs/2102.09756 (2021). arXiv:2102.09756 `https://arxiv.org/abs/2102.09756`

[220] Meng Wu, Haibo Zeng, and Chao Wang. 2016. Synthesizing Runtime Enforcer of Safety Properties Under Burst Error. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9690)*. Springer, 65–81. `https://doi.org/10.1007/978-3-319-40648-0_6`

[221] Yuhuai Wu, Albert Jiang, Jimmy Ba, and Roger Baker Grosse. 2021. INT: An Inequality Benchmark for Evaluating Generalization in Theorem Proving. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. `https://openreview.net/forum?id=O6LPudowNQm`

[222] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). arXiv:1609.08144 `http://arxiv.org/abs/1609.08144`

[223] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Neural Networks Learn. Syst.* 32, 1 (2021), 4–24. `https://doi.org/10.1109/TNNLS.2020.2978386`

[224] Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. In *Proceedings of the 36th International Conference*

on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97). PMLR, 6984–6994. `http://proceedings.mlr.press/v97/yang19a.html`

[225] Steve Zdancewic and Andrew C. Myers. 2003. Observational Determinism for Concurrent Program Security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*. IEEE Computer Society, 29. `https://doi.org/10.1109/CSFW.2003.1212703`

[226] Rodger E Zeimer and WH Tranter. 1995. Principles of communications: systems, modulation, and noise. `https://doi.org/10.1002/oca.4660080208`