Logical Methods for the Hierarchy of Hyperlogics

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

Jana Hofmann

Saarbrücken, 2022

Abstract

In this thesis, we develop logical methods for reasoning about hyperproperties. Hyperproperties describe relations between multiple executions of a system. Unlike trace properties, hyperproperties comprise relational properties like noninterference, symmetry, and robustness. While trace properties have been studied extensively, hyperproperties form a relatively new concept that is far from fully understood. We study the expressiveness of various hyperlogics and develop algorithms for their satisfiability and synthesis problems.

In the first part, we explore the landscape of hyperlogics based on temporal logics, first-order and second-order logics, and logics with team semantics. We establish that first-order/second-order and temporal hyperlogics span a hierarchy of expressiveness, whereas team logics constitute a radically different way of specifying hyperproperties. Furthermore, we introduce the notion of temporal safety and liveness, from which we obtain fragments of HyperLTL (the most prominent hyperlogic) with a simpler satisfiability problem.

In the second part, we develop logics and algorithms for the synthesis of smart contracts. We introduce two extensions of temporal stream logic to express (hyper)properties of infinite-state systems. We study the realizability problem of these logics and define approximations of the problem in LTL and HyperLTL. Based on these approximations, we develop algorithms to construct smart contracts directly from their specifications.

Zusammenfassung

In dieser Arbeit beschreiben wir logische Methoden, um über Hypereigenschaften zu argumentieren. Hypereigenschaften beschreiben Relationen zwischen mehreren Ausführungen eines Systems. Anders als pfadbasierte Eigenschaften können Hypereigenschaften relationale Eigenschaften wie Symmetrie, Robustheit und die Abwesenheit von Informationsfluss ausdrücken. Während pfadbasierte Eigenschaften in den letzten Jahrzehnten ausführlich erforscht wurden, sind Hypereigenschaften ein relativ neues Konzept, das wir noch nicht vollständig verstehen. Wir untersuchen die Ausdrucksmächtigkeit verschiedener Hyperlogiken und entwickeln ausführbare Algorithmen, um deren Erfüllbarkeits- und Syntheseproblem zu lösen.

Im ersten Teil erforschen wir die Landschaft der Hyperlogiken basierend auf temporalen Logiken, Logiken erster und zweiter Ordnung und Logiken mit Teamsemantik. Wir stellen fest, dass temporale Logiken und Logiken erster und zweiter Ordnung eine Hierarchie an Ausdrucksmächtigkeit aufspannen. Teamlogiken hingegen spezifieren Hypereigenschaften auf eine radikal andere Art. Wir führen außerdem das Konzept von temporalen Sicherheits- und Lebendigkeitseigenschaften ein, durch die Fragmente der bedeutensten Logik HyperLTL entstehen, für die das Erfüllbarkeitsproblem einfacher ist.

Im zweiten Teil entwickeln wir Logiken und Algorithmen für die Synthese digitaler Verträge. Wir führen zwei Erweiterungen temporaler Stromlogik ein, um (Hyper)eigenschaften in unendlichen Systemen auszudrücken. Wir untersuchen das Realisierungsproblem dieser Logiken und definieren Approximationen des Problems in LTL und HyperLTL. Basierend auf diesen Approximationen entwickeln wir Algorithmen, die digitale Verträge direkt aus einer Spezifikation erstellen.

Acknowledgements

It's been nine years since I moved to Saarbrücken to study Computer Science. I would never have thought back then that I would stay for so long. The fact that I did is due to the amazing people that make this place so absolutely wonderful. Their passion for research (and teaching!) is truly inspiring and made me feel welcome from the very first semester of my Bachelor's studies. My very first thanks, therefore, goes to everyone who dedicates so much of their time, energy, and love to this campus.

I am incredibly grateful to Bernd for allowing me to be part of his group and guiding me through this journey. It always felt as if you knew exactly what kind of guidance I needed. Your door was open whenever I needed advice or to discuss some proof, but I never felt pressured in any way. Thank you for placing your complete trust in my abilities and for valuing my ideas (even at times when these ideas were clearly not that valuable). The many things you taught me about research and academic life couldn't be more valuable. You also knew when to let me spread my wings and go off exploring to find out what excites me.

My amazing colleagues are the main reason I enjoyed this journey so much and why I really missed going to the office during covid lockdowns. Alex, Chris, Christa, Felix, Florian, Frederik, Hadar, Hazem, Jan, Jesko, Julian, Leander, Malte, Martin, Max, Michael, Mouhammad, Niklas, Noemi, Norine, Peter, Raven, Sabine, Swen: thank you for sharing my excitement, for complaining with me about Reviewer 2, for your honest feedback, for pre-deadline dinners with Indian takeout, for many good laughs, for (too) many coffee breaks, and in general for being such nice people. Having been part of the Reactive Systems Group makes me feel like the luckiest person. Special thanks go to Chris, my office mate and good friend throughout the years. Thank you for walking the path with me right from the first semester until now. You taught me how to make an introduction sound exciting; I hope this work lives up to your standards.

If there's one thing I've learned during this Ph.D., it's that research really is teamwork. To my co-authors, Bernd, Chris, David, Fan, Florian, Noemi, Norine, Markus, Jonni, Juha, Julia, Leander, Raven, and Yannick: thank you for many lively discussions, for sharing your perspectives, and for convincing me when mine were wrong. I really enjoyed our projects. I would also like to thank Chris, Hazem, Noemi, Niklas, Raven, and Yannick for their feedback on this thesis; for finding typos, and pointing out all the things that needed polishing. Thank you for your valuable time. This also applies to Dave and Juha, who kindly agreed to review this thesis.

To the whole 2020/21 Programming I team: it was an absolute pleasure to go on this adventure with you; I still cannot believe we managed to organize an introductory lecture with 600 students in the midst of a pandemic and still receive so much positive feedback. I learned a lot during that time (and not just about Discord bots). Another of my favorite teaching experiences was the re-invention of our maths precourse. Huge thanks to everyone who was part of that; it was quite a journey. I also wouldn't want to miss the many Wednesday evenings at Al Bacio's.

I was fortunate to share my time in Saarbrücken with an inspiring group of friends who made the city truly feel like home. Caro, Chris, Clara, Jesko, Kathrin, Nathalie, Noemi, Norine, Sebastian, and Yannick: thank you for these many evenings alternating between food from Café Bali, a good movie, port wine, watching soccer, and cooking together. Thank you for making me laugh, for listening, and for discussing all the small and big things in life. To our Mädelsrunde: you make me feel like being a female computer science researcher is the most normal thing in the world. I am excited to follow your future career paths and will always be cheering for you. Thank you, Sebastian, for watching all those Bond movies with me during the second covid lockdown. This was a busy and sometimes lonely period in which our shared evenings were something to look forward to. Liebe Becce, Kiki, Petti und Susi: Wir kennen uns nun schon weit mehr als unser halbes Leben lang. Danke, dass ihr mich immer noch versteht, auch wenn wir mittlerweile so unterschiedliche Leben führen.

No one knows and full-heartedly supports me like my family. Danke, Kira, dass du immer da bist, dass du mir so ähnlich und doch gerade richtig anders bist, um mir ein Vorbild zu sein. Danke, Mama und Papa, dass ihr mir beigebracht habt, meine Ziele zu verfolgen und an mich zu glauben. Danke, dass euer Zuhause so einfach wieder mein Zuhause werden konnte. Eurer Interesse, eure Unterstützung und eure Liebe haben mich immer begleitet und diesen Weg erst möglich gemacht.

Contents

Chapter 1

Introduction

In the last decades, the computer has changed our society at an astonishing pace.[1] Most critical infrastructures, such as communication, transportation, and financial technology, are now built on software systems. With the rise of autonomous systems, this trend will only intensify. In such complex systems, errors are hard to find and can have severe consequences, especially when they occur in the aviation industry [145] or affect our health system [75]. In the financial sector, millions of dollars have been stolen in recent years due to bugs in the implementation of blockchain-based cryptocurrencies and smart contracts [192, 163]. To increase the robustness of such critical systems and secure them against malicious attacks, their development must be based on rigorous mathematical foundations.

Formal methods is a branch of computer science that provides techniques for the development of systems that *provably* satisfy certain properties. As natural language cannot unambiguously express such properties, formal methods are based on *logics*, that is, mathematical specification languages. Many properties of concern are *temporal properties*, which describe the behavior of a system over time. Logics for the expression of temporal properties form the cornerstone of the formal methods field. Algorithms based on temporal logics revolutionized the hardware design process in the 1990ies and early 2000s [142, 105, 200]. Nowadays, temporal logics are employed routinely by companies such Amazon Web Services and Microsoft [185, 160]. In recent years, formal methods have also found increasing application for the verification of smart contracts [118, 188, 1], which led to the creation of various start-ups. The success of temporal logic is the result of decades of foundational research. There exists a multitude of different logics, whose relative expressiveness is well understood. Algorithms based on symbolic methods or on the connection of temporal logics to automata can now automatically verify systems with millions of states [38].

[1]Here, former IBM CEO Thomas J. Watson comes to mind, who in 1943 allegedly said that "there is a world market for maybe five computers". Even though representative of its time, this statement seems to be an urban myth [214].

The expressiveness of temporal logics is, by design, limited to reasoning about single system executions. The need for more expressive logics became particularly clear after the discovery of the Meltdown [166] and Spectre [149] attacks in 2017. They show that an attacker can obtain secret information by observing differences between multiple CPU executions. To specify the absence of such a vulnerability, we need *relational properties*, i.e., properties that compare multiple executions of a system. Similar properties can be found in various branches of computer science; further examples are robustness [69] and fairness properties like symmetry [94]. In 2010, Clarkson and Schneider introduced *hyperproperties* [51] as a unifying notion for this type of properties. Compared to standard temporal properties, the research on hyperproperties is still in its infancy. The fact that hyperproperties relate multiple system executions makes the development of a comprehensive theory a challenging task. We still lack a deep understanding of the various ways to specify hyperproperties. We also lack the variety of logical reasoning methods that we now have for standard temporal logics.

In this thesis, we advance the logical foundations for hyperproperties. We explore the expressiveness of different types of hyperlogics, design novel logics, and develop algorithms for logical reasoning. We do so in two parts.

In the first part, we advance our understanding of existing logics for hyperproperties. We systematically study the relative expressiveness of hyperlogics built from different types of standard (non-relational) logics. We consider temporal logics, first-order (FO) and second-order (SO) logics, and logics with team semantics. We show that hyperlogics based on temporal and FO/SO logics form a strictly ordered hierarchy of expressiveness. Hyperlogics based on team semantics, on the other hand, constitute a new perspective on hyperproperties and are difficult to compare to other hyperlogics. Furthermore, we investigate the satisfiability problem of HyperLTL, the most prominent logic in the hierarchy. The satisfiability problem asks if a specification is satisfied by any system at all; it can be used as a means of analysis, e.g., to find implications between specifications. We introduce fragments of HyperLTL that have a simpler satisfiability problem and provide algorithmic approximations of the problem.

In the second part, we describe novel logics to construct feasible algorithms for the synthesis problem. The synthesis problem is the task of automatically constructing a system from a specification; it is extremely intriguing but known to be computationally hard, especially for hyperproperties. Our goal is to synthesize smart contracts, which are a prime example of the necessity of developing software systems based on rigorous foundations. We first describe the synthesis of smart contracts from a functional specification, for which we equip temporal stream logic (TSL) with universally quantified parameters. In a second step, we extend the approach to hyperproperties specified in the novel logic HyperTSL. While we explicitly aim to develop logical methods for smart contracts, the applicability of our techniques is not limited to smart contracts but can be extended to other software and infinite-state systems.

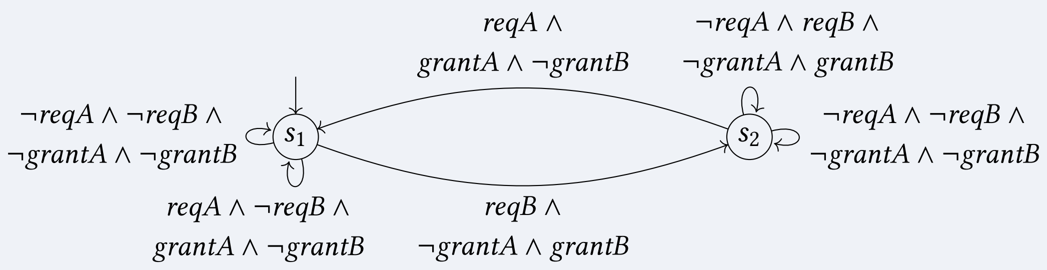
This thesis builds on a range of concepts, on which we elaborate in the following sections. We first provide background on reactive systems and temporal logics as well as hyperproperties. We further discuss the satisfiability and synthesis problem and give a quick introduction to smart contracts. Finally, we present the main contributions of this thesis in more detail and provide a list of supporting publications.

1.1 Reactive Systems and Temporal Logics

The goal of formal methods is to develop systems together with proofs of their correctness. One such method is *verification*, which proves that an existing system is correct. Another is *synthesis*, which constructs a system directly from a formal specification. Other approaches concern the analysis of the specification, for example the *satisfiability* problem. For any of these methods, we must 1) decide how to represent the system with an abstract model and 2) describe its properties in a suitable logic.

Modeling Reactive Systems. This thesis is concerned with *reactive systems* [131], which are systems that run for an indefinite amount of time and interact continuously with their environment. Typical reactive systems are cyber-physical systems, which connect digital systems with the physical world, for example in autonomous vehicles. Also software, for example smart contracts, are often reactive systems. There are various ways to model reactive systems. The standard model are *transition systems*, which consist of states and transitions, which describe how the systems reacts to inputs.

Example 1.1. The transition system below is one of the classic examples from the literature. It models a system in which two agents (for example, two processes) both need access to a shared resource (for example, a memory location). The transition system implements a scheduler that decides which agent gets access depending on the requests it receives. We use the Boolean proposition *reqX* to state that agent *X* requests access. *grantX* denotes that agent *X* is granted access by the scheduler.


$reqA \land$ $grantA \land \neg grantB$
$\neg reqA \land reqB \land$ $\neg grantA \land grantB$
$\neg reqA \land \neg reqB \land$ $\neg grantA \land \neg grantB$
s_1
s_2
$\neg reqA \land \neg reqB \land$ $\neg grantA \land \neg grantB$
$reqA \land \neg reqB \land$ $grantA \land \neg grantB$
$reqB \land$ $\neg grantA \land grantB$

The two states of the system memorize which of the agents was the last to get access to the resource. Like that, if both agents request access, the agent that was *not* the last to use the resource will get access.

Logics for Temporal Properties. The properties we are interested in are *temporal properties*, which describe the behavior of the system over time. Typical temporal properties are *safety* and *liveness properties* [158, 8, 7]. Safety properties describe what the system is *not* supposed to do, for instance, “the robot never enters the safety-critical area of the production hall” or “the server cannot be accessed if the user did not enter the correct password beforehand”. Liveness properties, on the other hand, describe how to satisfy a specification in the future, for example “the car eventually reaches its destination”. A class of logics that describe temporal properties are called *temporal logics* and originate in the work of Arthur Prior [164]. The most successful temporal logic is linear temporal logic (LTL) [190], for which, among other contributions to formal methods, Amir Pnueli received the Turing award in 1996. LTL combines Boolean connectives such as “and” $\wedge$ and “not” $\neg$ with temporal modalities that reason about the order of events produced by a system.

Example 1.2. For the transition system depicted in Example 1.1, the following LTL formula expresses the safety property that no grant is given spuriously when no agent requests it.

$$\square(\neg reqA \wedge \neg reqB \rightarrow \neg grantA \wedge \neg grantB)$$

Above, the $\square$ operator states that a formula has to hold at all times. A typical liveness property is the statement that every request by agent A is eventually ($\diamond$) answered with a grant.

$$\square(reqA \rightarrow \diamond\, grantA)$$

Based on LTL, a range of other temporal logics have been proposed. Most of them either take the *linear-time view* or the the *branching-time view* of a system. Linear-time logics like LTL argue about the different execution traces of a system, while branching-time logics like computation tree logic (CTL*) [74] see the system executions as a tree, where every branch stands for a possible next step of the system. Branching-time logics thus describe the potential nondeterminism in the system. Which view is the better one to model time is a long-standing debate in the field of formal methods [221].

Temporal properties are not only expressible in temporal logics but also in other types of logics like first-order and second-order logics. Most notably, *Kamp’s Theorem* [144], as interpreted by Gabbay, Pnueli, Shelah, and Stavi [109], states that LTL can express the same properties as first-order monadic logic of order (FO[<]). FO[<] is a first-order logic that is interpreted over the natural numbers $\mathbb{N}$. It has only monadic (unary) predicates except for the interpreted $<$ relation.

Since the proposal of LTL in the 1970ies, temporal logics have been widely adopted in industry (see, e.g., [223] for an overview). Due to the comparatively small size of a circuit, the first applications of temporal logics were in hardware verification, for example at Intel [97]. In the 2000s, there have been first efforts to standardize temporal specification languages [155]. The advancements of large-scale verification techniques based on symbolic methods [38, 49, 222] paved the path for an increasing application of formal methods to software as well [14]. Nowadays, temporal logics like TLA+ [159] are used regularly in industry, for example at Microsoft [161] or Amazon Web Services [185].

1.2 Hyperproperties

Hyperproperties describe relations between multiple execution traces of a system. A prominent example is noninterference [116], which states that the output some normal, low-security user receives from a system may not depend on the initial value of some high-security variable in the system. Dependency is a hyperproperty: if the user provides the system with the same input, then it must receive the same output, even if the value of the secret variable is different. A general, set-based definition of hyperproperties was introduced in 2010 by Clarkson and Schneider [51]. It plainly defines hyperproperties as sets of sets of traces, as opposed to temporal trace properties (as defined by LTL), which are sets of traces. Before the introduction of the general term, hyperproperties haven been studied mostly individually in various branches of computer science, especially in information flow security and privacy research.

Information Flow Security. Information flow security reasons about the flow of sensitive data in a system. Besides noninterference, a typical property is observational determinism [233], which expresses that observing the same sequence of low-security inputs leads to observing the same low-security outputs. A prominent privacy property is differential privacy, which requires that similar inputs lead to a limited amount of change in the outputs [72]. Sometimes, flow of information cannot be prevented or is even desired. Declassification properties specify when and how much information is allowed to become public [202]. All the above properties are relational properties. For information flow policies, most work focuses on the verification problem, see, for example, [23] for an overview. Many information flow policies are implicitly universally quantified, that is, they relate any k traces of the system. This makes it possible to reduce the verification problem to verifying standard, non-relational properties. One of the major approaches is to perform a self-composition [18, 63], which constructs the product of k instances of the same system, and then applies classic verification methods on the resulting system. Some variant of self-composition can be found at the heart of many approaches for the verification of hyperproperties, even beyond purely universally quantified hyperproperties [94, 77, 27, 59]. Another verification approach is via *relational Hoare logic* [24, 106], which lifts standard Hoare logic to reasoning about two programs at the same time. Relational Hoare logics have also been extended to probabilistic systems [19], quantitative reasoning [20], and k-safety hyperproperties [213]. For an overview over relational verification based on Hoare logics see [183]. There exists numerous other approaches for the verification of information flow policies, e.g. static approaches via abstract interpretation [12, 127, 45], dynamic methods [234, 65, 44], and combinations of both [201].

Relational properties received considerable attention after the discovery of the Meltdown [166] and Spectre [149] attacks, which revealed that most modern processors were amenable to sophisticated attacks that exploit the effects of speculative executions at the microarchitectural level. As a result, various methods and tools were proposed to detect and prevent such attacks (e.g., [226, 11, 68]). The core problem of Meltdown and Spectre is that speculatively loaded data can be leaked to an attacker through cache timing side channels. A side channel attack is relational: to learn what data was loaded into the cache, the attacker compares the access time to certain memory locations. The absence of side channels can therefore only be ensured by checking relational properties, for example *speculative noninterference* [119].

Beyond Information Flow Security. The interest in hyperproperties has for a long time been driven by information flow policies, but over time it became apparent that many properties outside the privacy area are hyperproperties as well. Examples are robustness [69] ("if the system receives similar inputs, it produces similar outputs") and fairness properties like symmetry [94] ("the system treats different users symmetrically if they act symmetrically"). Many epistemic properties are hyperproperties as well. Epistemic properties argue about the knowledge of agents in distributed systems and have been studied long before the term "hyperproperties" was coined [76, 126, 125]. Some epistemic logics, for example LTL extended with the *knowledge operator* [126], can be captured by logics for hyperproperties [195]. Epistemic logics themselves have also been applied to specify information flow policies like noninterference and declassification [15]. Recently, some instances of counterfactual causality in reactive systems have been interpreted as hyperproperties [54, 53]. Counterfactual reasoning tries to determine the causes that led to some event, for example what inputs are the reason for the observed outputs.

Logics for Hyperproperties. Since the introduction of hyperproperties, various logics for the expression of hyperproperties have been introduced, most of which build on existing non-relational logics. The most prominent hyperlogics is HyperLTL [50], which is an extension of LTL. HyperLTL uses the syntax of LTL and adds prefixed quantifiers $\forall$ and $\exists$ to quantify traces from the set of executions a system can produce. The atomic propositions of the inner LTL formula are indexed with the quantified trace variables to denote what trace they refer to.

Example 1.3. Symmetry in the scheduling system depicted in Example 1.1 would mean that neither process A nor process B are favored over the other by the scheduler. One way to express symmetry in HyperLTL is the following formula.

$$\begin{aligned}&\forall \pi. \forall \pi'. \square((reqA_{\pi} \leftrightarrow reqB_{\pi'}) \wedge (reqB_{\pi} \leftrightarrow reqA_{\pi'}))\\ &\qquad \rightarrow \square((grantA_{\pi} \leftrightarrow grantB_{\pi'}) \wedge (grantB_{\pi} \leftrightarrow grantA_{\pi'}))\end{aligned}$$

The formula quantifies two execution traces π and π' of the system and states that if these two traces have the requests of agents A and B exactly swapped, than also the grants for A and B must be swapped.

Based on HyperLTL, various algorithms for logical reasoning have been proposed, tackling, for example, the verification [94, 59, 138], satisfiability [173, 82, 98], runtime verification [5, 88, 57], and synthesis problems [86, 85]. The common algorithmic challenge of all these problems is to deal with quantifier alternations ($\forall\exists$ or $\exists\forall$ formulas), which prevent direct reductions to the respective LTL problem. In Section 1.3, we discuss in more detail the satisfiability and synthesis problem of HyperLTL.

By extending HyperLTL with additional features, more expressive logics have been obtained, for example, for quantitative hyperproperties [89, 204] and hyperproperties in asynchronous [122, 22, 35] and probabilistic systems [3, 2, 22]. Other hyperlogics do not build on LTL but on other temporal logics, resulting, for instance, in HyperCTL* [50] and HyperQPTL [195, 55].

While most hyperlogics are temporal logics, there also exist extensions of other logical formalisms to hyperproperties. FO[$<$], for example, has been equipped with an additional binary predicate, the equal-level predicate, resulting in the hyperlogic FO[$<, E$]. The equal-level predicate relates points on different execution traces and thereby enables the logic to express hyperproperties. Interestingly, while LTL and FO[$<$] are expressively equivalent, HyperLTL and FO[$<, E$] are not [95]. Another type of hyperlogic has been obtained by interpreting LTL, CTL, and CTL* under team semantics [153, 120]. Team semantics provide an alternative to the classic Tarski semantics of logics. In team semantics, formulas are not interpreted with respect to a single assignment of variables but with respect to a set of assignments. The initial idea behind team semantics was to define compositional logics that can easily express (in)dependence statements [137, 220]. It was later observed that temporal team logics are actually hyperlogics [153]. The fact that they do not employ explicit quantification over traces or variables makes temporal team logics radically different from other temporal hyperlogics. Indeed, the expressiveness of TeamLTL is incomparable to that of HyperLTL [153].

1.3 Satisfiability and Synthesis

The study of logics is not an end in itself, the ultimate goal of formal methods is to construct systems which, formally proven, satisfy the properties described in a logic. In this thesis, we study the satisfiability problem as well as the synthesis problem.

Satisfiability. The satisfiability (SAT) question is one of the standard problems of logics. For a given formula, it asks if there is any model that satisfies the formula. In its negation, the problem can be formulated as "does the formula hold trivially on every system?". One application of SAT solving is as a preprocessing step to sanity-check the properties one wants to enforce on a system. If a formula is unsatisfiable, the specification probably does not express what was intended. With satisfiability solving, it also possible to detect if two formulas imply each other, i.e., if one is the stronger formulation of the other. Apart from formula preprocessing, satisfiability solvers are hugely successful to encode and solve more complex problems both in academic and industrial applications. SAT solvers are used, for example, for model checking [29, 177] and planning [113] problems, but also to resolve dependencies in custom package installers [217], or even to analyze the effect of genes to diseases [170]. Even if the Boolean SAT problem is famously NP-complete [60, 165], modern SAT solvers solve formulas with hundreds of thousands of variables and millions of clauses [16]. There is very active research on Boolean SAT solving: in 2021, over 50 tools competed in the annual SAT solving competition [17].

The more expressive a logic is, the harder is its satisfiability problem. LTL satisfiability solving is PSPACE complete [211]. For HyperLTL, the problem is PSPACE-complete for formulas with a $\exists^*\forall^*$ quantifier prefix but undecidable in general [82]. More precisely, the general problem sits in Σ_1^1 and is thus highly undecidable, meaning that it cannot be encoded in first-order logic [98]. The tool EAHyper [87] is a SAT solver for the decidable fragment of HyperLTL. It reduces the problem to LTL satisfiability checking. For the undecidable fragment, the only algorithmic approach is provided by the tool MGHyper [83], which implements a search for satisfying finite trace sets of increasingly larger size.

Synthesis. The reactive synthesis problem, which dates back to Alonzo Church in 1953 [46], is considered the holy grail of formal methods. Its goal is to automatically construct a reactive system (i.e., a transition system) from a given logical specification. True to the motto "describe *what* to do, not *how* to do it", the ultimate goal of synthesis is to replace programming as we know it. The reactive synthesis problem can be understood as a game between the environment (consisting of, for example, some user input or sensor data) and the system to be synthesized. The environment produces an input to which the system has to answer with an output. The environment produces the next input in response, and so on and so forth. This game goes on (in theory) forever and if the resulting execution trace satisfies the specification, the system player wins. The synthesis task is thus to construct a *strategy* of how to choose the outputs such that the system wins for every possible sequence of environment inputs. A finite representation of this strategy as a transition system then implements the system.

Synthesis is an intriguing idea but turned out to be notoriously hard to solve. For LTL, for example, the problem is known to be 2EXPTIME-complete [191] in the size of the formula. First solutions for the synthesis problem are based on the work of Büchi and Landweber [37] and proceed by explicit game solving. The formula is first translated into an automaton, in the case of LTL into a nondeterministic Büchi automaton [224]. In the next step, the automaton is determinized [203] and interpreted as a game, where in every state, either the system player or the environment player can choose the next move, and system states and environment states alternate. Solving the game then determines if there is a winning strategy for the system player [191]. If this is the case, the specification is *realizable*.

Since Church first formulated the synthesis problem, there has been considerable progress on the matter. Explicit methods based on parity game solving are combined with a range of clever algorithmic ideas that decompose the LTL formula and construct only the parts of the game that are needed to determine the winner of the game [169]. Inspired by the success for the verification task, symbolic methods group the states of the game by encoding them with complex data structures, e.g., by representing the transition function as a BDD [79, 73]. Bounded synthesis interprets the synthesis problem as a search problem by searching for a smallest implementation that satisfies the formula [207, 78]. Other approaches simplify the synthesis problem by restricting the type of formula, most successful in form of the GR(1) fragment, which assumes that the LTL formula is given as a combination of assumptions and guarantees [189]. For an overview over the history of the synthesis problem, see, for example, [80].

The progress on synthesis from LTL specifications opens the door to solving the task for more expressive logics. A recent extension of LTL is *temporal stream logic* (TSL) [93], which extends LTL's atomic propositions with cells that hold data from a possibly infinite domain. The cell mechanism is combined with uninterpreted functions and predicates. Functions modify the cells whereas predicates can be used to query properties of the contents of the cells. The synthesis question for TSL asks if there is a strategy that satisfies the formula for every interpretation of the functions and predicates. While TSL synthesis is in general undecidable, the problem can be soundly approximated via LTL synthesis [93]. TSL has been successfully applied to specify and synthesize an arcade shooter game running on an FPGA [112] and functional reactive programs [92].

Example 1.4. With TSL, we can express, for example, that the scheduler should count how often each agent requested access to the shared resource. To do so, we introduce two cells `requestsA` and `requestsB` and require that their value is incremented with every request. If there is no request from the agent, the value of the cell should stay the same. For agent A, this can be formulated as

$$\begin{gathered}\square((\mathit{reqA} \rightarrow [\![\mathtt{requestsA} \leftarrowtail \mathtt{requestsA} + 1]\!]) \wedge \\ (\neg \mathit{reqA} \rightarrow [\![\mathtt{requestsA} \leftarrowtail \mathtt{requestsA}]\!]))\end{gathered}$$

Now, if both agents request access to the resource, the agent which accumulated more requests so far should get access.

$$\square(\mathit{reqA} \wedge \mathit{reqB} \wedge \mathtt{requestsA} > \mathtt{requestsB} \rightarrow \mathit{grantA})$$

Synthesis from hyperproperties is even harder than synthesis from standard temporal properties. So far, the synthesis has only been studied for HyperLTL, for which the problem is undecidable already for formulas with two or more $\forall$ quantifiers [86]. This excludes most of the relevant information flow policies, which often fall into the $\forall^2$ fragment. As a remedy, there exist an implementation of bounded synthesis for $\forall^*$ HyperLTL formulas [86].

1.4 Smart Contracts

Cryptocurrencies, which are digital coins implemented on top of the blockchain, have recently received a considerable amount of attention. The initial promise – making traditional banks superfluous by employing distributed consensus algorithms – is faced with frequent attacks and scams, in which millions of dollars are lost [192, 163]. Digital coins and, more generally, smart contracts are thus a prime example why the development of security-critical systems needs to be based on solid mathematical reasoning.

Blockchains. The underlying technology of smart contracts has, despite the recent headlines, great potential. Smart contracts are small programs that digitally implement an agreement between multiple parties. These are mostly contracts in which a certain order of transactions needs to be observed, for example when selling or auctioning goods. Other smart contract implement digital coins or token systems. The slogan of smart contracts is “code is law” meaning that a contract written in a programming language unambiguously determines the order of transactions – and not a classical, potentially ambiguous, contract written in natural language. Smart contracts are deployed on the blockchain. Blockchains are distributed ledgers, or, simply speaking, a list of blocks with data. The key idea is that everyone in the distributed network agrees on the current state of the blockchain. Whenever a new block needs to be appended to the chain, a *consensus algorithm* is executed to ensure that everybody in the network agrees on whether or not the block was successfully appended. Due to the consensus mechanism, every successful interaction with the contract is irrefutably stored on the blockchain. This has several consequences. First, there is no need for a trusted third party watching the correct execution of the contract. Second, as the contract itself is data on the blockchain, contracts cannot by altered once deployed. This produces trust but also means that contracts cannot be updated, for example, to fix a bug.

Ethereum Smart Contracts. The most popular blockchain for smart contracts is the Ethereum blockchain [101]. Known Ethereum smart contracts are, for example, token systems like non-fungible tokens (NFTs), which prove ownership over a specific item like a piece of art. Most Ethereum smart contracts are implemented in Solidity [102], an object-oriented programming language with influences from C++, Javascript, and Python. Solidity also has some custom features like the possibility to rollback all state changes made by the current function call. Another feature is the possibility to send Ether, Ethereum's native currency, between different contract addresses. Solidity smart contracts are compiled to Bytecode that is executed on the Ethereum Virtual Machine (EVM). As a change of state needs to validated on the blockchain, every transaction with the contract costs gas, for which users commonly pay in Ether.

Formal Methods for Smart Contracts. Fueled by the many bugs that led to a decrease in trust, there is very active research on formal foundations for smart contracts. Some efforts have been invested to formally describe the semantics of (parts of) Solidity and the EVM [117, 141, 124, 28]. Other approaches propose new languages with a formal semantics or build-in state machines [52, 208, 1]. Formal methods for smart contracts are enabled by the fact that the programs implementing the contract are usually quite small. We focus on the *temporal control flow* for smart contracts, which specifies the order of transactions that needs to be followed. There is a number of successful approaches to model check smart contracts against linear-time as well as branching time temporal logics [188, 215, 184]. Another line of work models the temporal control flow of smart contracts with a transition system, which is then checked against the implementation [227, 157] or automatically translated to Solidity code [174, 175].

1.5 Contributions

This thesis advances the logical foundations of hyperproperties. We examine the expressiveness of existing and novel logics for hyperproperties and design algorithms to solve the satisfiability and synthesis problem of hyperlogics.

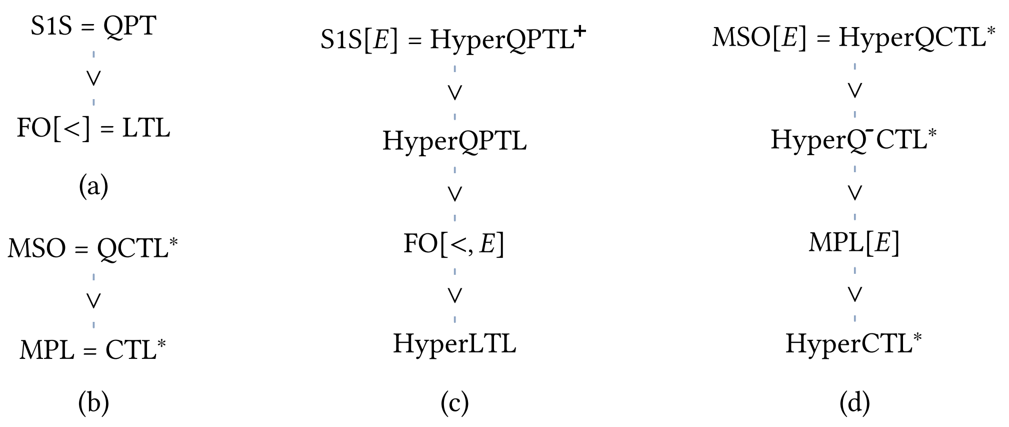


Figure 1.1: The linear-time hierarchies of standard logics (a) and hyperlogics (c), and the branching-time hierarchies of standard logics (b) and hyperlogics (d).

1.5.1 Expressiveness of Hyperlogics

The main challenge when expressing hyperproperties lies, as compared to trace properties, in the additional dimension. Trace logics reason over one-dimensional traces. Hyperlogics, on the other hand, reason over sets of traces and thus have a two-dimensional domain. This opens up a multitude of design decisions. Understanding the impact of these decisions on the expressiveness of a logic is crucial to learn how we can best express hyperproperties in different systems. We study a broad range of hyperlogics built from different types of base logics. Our aim is to understand how these different features influence the expressiveness of the logics.

A Hierarchy of Hyperlogics. We explore the relative expressiveness of three different types of hyperlogics: quantifier-based temporal hyperlogics, hyperlogics based on first-order and second-order logics, and temporal team logics. Inspired by Kamp's theorem, we first examine quantifier-based temporal hyperlogics in comparison to FO/SO hyperlogics. To do so systematically, we interpret the addition of trace/path quantifiers to temporal logics and the equal-level predicate to FO/SO logics as general "recipes" to obtain a hyperlogic. We apply these recipes to those temporal logics and FO/SO logics that are known to be expressively equivalent in the standard hierarchy of logics. As it turns out, the equal-level predicate for FO/SO logics consistently generates more expressive logics than trace and path quantifiers do for temporal logics. This results in two hierarchies of expressiveness, as depicted in Figure 1.1. Our study also underlines that lifting a logic to a hyperlogic is not always straight-forward. Subtle changes (e.g., in the definition of HyperQPTL) might lead to significant jumps in expressiveness.

As a second step, we attempt to include hyperlogics obtained by interpreting LTL under team semantics into the hierarchy of linear-time hyperlogics. We define two extensions of TeamLTL that can express two general classes of LTL-definable Boolean relations. We discover that it is much harder to relate the expressiveness of these logics to quantifier-based temporal hyperlogics than it is for FO/SO hyperlogics. This is because TeamLTL and its extensions do not use explicit quantification over the set of traces. Instead, they reason about subsets by repeatedly splitting the set in two. Nevertheless, we show that both logics can be captured by HyperQPTL^{+}. We also define an expressive fragment of one of the logics, which is still incomparable to HyperLTL but subsumed by HyperQPTL.

Extensions of TSL. As a concrete application of temporal logics, we examine temporal (hyper)properties occurring in the context of smart contracts. Our goal is to specify a contract's implicit control flow graph. Smart contracts are software systems, so we need logics that allow for precise specifications in the presence of variables that hold data from infinite domains. We define extensions of TSL to accurately capture the temporal properties of a smart contract. The first extension equips TSL with universally quantified parameters. Using this logic, we can, for example, express that in an election, each address may only vote once.

$$\forall \mathtt{m}. \square(\mathtt{vote}(\mathtt{m}) \rightarrow \bigcirc \square \neg \mathtt{vote}(\mathtt{m}))$$

Above, `vote` models the method call and `m` is a parameter that refers to the sender of the message. As second extension, we define two hyperlogics based on TSL: HyperTSL and HyperTSL$_{\mathrm{rel}}$[2]. The logics are designed to express hyperproperties in infinite-state systems and software in particular. With HyperTSL, we can express, e.g., symmetry in voting protocols. For the case of an election with two candidates `A` and `B`, the following formula states that the winner is chosen symmetrically as long as the votes come in symmetrically on any two traces.

$$\begin{aligned}\forall \pi, \pi'. \big((\llbracket \mathtt{winner} \leftarrowtail \mathtt{A()} \rrbracket_{\pi} \leftrightarrow \llbracket \mathtt{winner} \leftarrowtail \mathtt{B()} \rrbracket_{\pi'})\\ \wedge\ (\llbracket \mathtt{winner} \leftarrowtail \mathtt{B()} \rrbracket_{\pi} \leftrightarrow \llbracket \mathtt{winner} \leftarrowtail \mathtt{A()} \rrbracket_{\pi'})\big)\\ \mathcal{W}\ \big((\mathtt{voteA}_{\pi} \nleftrightarrow \mathtt{voteB}_{\pi'}) \vee (\mathtt{voteB}_{\pi} \nleftrightarrow \mathtt{voteA}_{\pi'})\big)\end{aligned}$$

Above, `winner` is the field that stores who the current winner of the election is. Compared with HyperTSL, HyperTSL$_{\mathrm{rel}}$ predicates may range over several traces. We argue that this additional expressiveness makes the logic less suited as basis for the synthesis problem.

[2]Earlier versions of HyperTSL and HyperTSL$_{\mathrm{rel}}$ were proposed as part of Julia Tillman's Bachelor's thesis at Saarland University in 2020, which the author supervised together with Norine Coenen.

1.5.2 Algorithms for Hyperlogics

The development of formal methods based on hyperlogics is known to be a challenge. Problems like satisfiability and synthesis are undecidable already for HyperLTL, which is one of the least expressive logics we consider in this thesis. The question is thus how to still find solutions for these problems. We tackle this challenge from two directions. On the one hand, we search for fragments of the logics for which the problems become easier. This means either decidability or that we can reduce the problem to logics for which we have better algorithms. On the other hand, we define approximations of the problems in simpler logics. As long as we can guarantee soundness, these approximations often work well in our experiments.

Temporal Safety and Liveness for the Satisfiability of HyperLTL. We examine the satisfiability problem of $\forall^*\exists^*$ HyperLTL, which is in general Σ_1^1-complete. We introduce the semantic notion of temporal safety and temporal liveness as a novel way to obtain fragments of HyperLTL. We show that the complexity of the HyperLTL satisfiability problem drops to coRE-complete for temporal safety HyperLTL formulas. We obtain this result by a reduction to first-order logic, what means that semi-decision procedures like tableau become applicable. For temporal liveness, in contrast, the complexity stays the same, already for very simple formulas. As a remedy, we propose a sound approximation to find largest models for general $\forall\exists^*$ HyperLTL formulas. The algorithm can show satisfiability as well as unsatisfiability.

Synthesis of Smart Contract Control Flows. We present algorithms to synthesize the control flow of smart contracts from parameterized TSL and HyperTSL. In parameterized TSL, parameters are instantiated from an infinite domain; the implied system is thus an infinite-state system. Therefore, we need to find a way to (i) synthesize the system and (ii) to find a gas-saving, finite representation of the system in Solidity. We propose to first synthesize the finite system that results from any instantiation of the parameters. Then, we split the system into a distributed, hierarchical structure that represents the infinite-state system in a compact way.

For HyperTSL, we first show how to reduce the problem to HyperLTL synthesis for the $\forall^*$ fragment and to LTL satisfiability for the $\exists^*$ fragment. More practically, we describe a two-step approach to extend smart contract synthesis from TSL to HyperTSL. The first step introduces the notion of pseudo hyperproperties, which detects hyperproperties that can be described as trace properties. Second, we propose to split the synthesis to first synthesize the most general system from the trace properties and then refine the system according to the hyperproperties.

1.6 Publications

Our contributions are the result of joint work with various co-authors. This thesis is based on the following peer-reviewed publications.

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

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

[225] Jonni Virtema, Jana Hofmann, Bernd Finkbeiner, Juha Kontinen, and Fan Yang. **Linear-time Temporal Logic with Team Semantics: Expressivity and Complexity**. *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2021).*

[25] Raven Beutner, David Carral, Bernd Finkbeiner, Jana Hofmann, and Markus Krötzsch. **Deciding Hyperproperties Combined with Functional Specifications**. *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2022).*

[58] Norine Coenen, Bernd Finkbeiner, Jana Hofmann, and Julia Tillman. **Smart Contract Synthesis Modulo Hyperproperties**. *To appear at the 36th IEEE Computer Security Foundations Symposium (CSF 2023).*

Furthermore, the thesis contains material from the following publication, which is currently under review.

[91] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. **Reactive Synthesis of Smart Contract Control Flows**. *arXiv 2022.* URL: `https://arxiv.org/abs/2205.06039`.

1.7 Structure of This Thesis

We present our results in two parts. Each part starts with a chapter that establishes the required preliminaries for the respective part. The preliminaries of Part II (Chapter 6) build on the preliminaries for Part I. Apart from the preliminaries, both parts are sufficiently self-contained to be read independently. Each part concludes with a chapter on closely related work. Finally, in Chapter 10, we discuss the results of this thesis and elaborate on future work.

Part I: The Hierarchy of Hyperlogics. In the first part of the thesis, we explore the general landscape of hyperlogics. Chapter 3 presents the expressiveness hierarchy. We start with the strict hierarchy of linear-time temporal and FO/SO logics in Section 3.1. In Section 3.2, we establish the relationship of two TeamLTL variants to quantifier-based temporal hyperlogics. Subsequently, in Section 3.3, we discuss the hierarchy of branching-time temporal and FO/SO hyperlogics.

In Chapter 4, we present our contributions to the HyperLTL satisfiability problem. Section 4.1 introduces the notion of temporal safety and presents the associated results. Section 4.2 does the same of temporal liveness. Lastly, we present our algorithm to find largest models $\forall\exists^*$ HyperLTL formulas in Section 4.3.

Part II: Synthesizing Smart Contracts. In the second part of the thesis, we turn towards logics and algorithms specifically for smart contracts. In Chapter 7, we examine the synthesis from trace properties. Section 7.1 recaps the synthesis of smart contracts from TSL, which was presented in [91] but is not part of the contributions of thesis. We extend the approach with an analysis of the resulting state machine in Section 7.2. Next, in Section 7.3, we introduce TSL with parameters and show how smart contracts can be conveniently specified with this logic. Section 7.4 presents a synthesis algorithm that constructs smart contracts written in Solidity. We evaluate the resulting tool on a benchmark set of different contracts in Section 7.5.

In Chapter 8, we discuss smart contract synthesis from hyperproperties. We introduce HyperTSL and HyperTSL$_{\text{rel}}$ in Section 8.1 and show how to specify hyperproperties of smart contracts in Section 8.2. In Section 8.3, we formally study the synthesis problem of $\forall^*$ and $\exists^*$ HyperTSL. We define the notion of pseudo-hyperproperties in Section 8.4 and describe a sound check via HyperLTL SAT solving. Finally, in Section 8.5, we describe a repair-like approach for HyperTSL synthesis building on TSL synthesis. We evaluate the approach in Section 8.6.

Part I

The Hierarchy of Hyperlogics

Chapter 2

Preliminaries

In this chapter we define the necessary preliminaries for the first part of the thesis. We first introduce linear-time and branching-time properties, and hyperproperties. We then discuss several models of computation. In the first part of the thesis, we represent reactive systems with finite-state transition systems, which are labeled with subsets of atomic propositions. In proofs we also use counter machines and Turing machines. Finally, we define various temporal, first-order and second-order logics for the definition of properties and hyperproperties.

2.1 Properties

Traces. Let an alphabet Σ be given. An infinite sequence $t \in \Sigma^\omega$ is called a *trace*, a finite sequence $t \in \Sigma^*$ a *finite trace*. For a trace t and a natural number $i \geq 0$, we denote the i-th element of the trace by $t[i]$. We also use this notation for finite-length traces and other sequences like tuples, provided that i is not greater than the length of the sequence. For a natural number $j \geq i$, $t[i, j]$ denotes the sequence $t[i]t[i+1] \dots t[j-1]t[j]$. Moreover, for an infinite trace t, $t[i, \infty]$ denotes the infinite suffix of t starting at position i. For two traces t and t' over Σ and Σ', we define a zipping operation $\mathit{zip}(t, t') = (t[0], t'[0])(t[1], t'[1]) \dots$, which defines a trace over $\Sigma \times \Sigma'$. We call $u \in \Sigma^*$ a *prefix* of $t \in \Sigma^\omega \cup \Sigma^*$ (written $u \sqsubseteq t$) if for some n, $|u| = n$, $|t| \geq n$, and $u[0, n] = t[0, n]$.

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* $\{s' \mid s' < s\}$ is totally ordered. We say that s' is the *child node* of s, if $s < s'$, and there is no s'' such that $s < s'' < s'$. A Σ*-labeled tree* $(\mathcal{T}, L)$ is a tree $\mathcal{T}$ equipped with a labeling function $L : S \to \Sigma$, which labels every node of $\mathcal{T}$ with an element from Σ. A *path* through a tree T is an infinite sequence

$p = s_0, s_1, \ldots$ such that for all s_i, s_{i+1}, node s_{i+1} is the child node of s_i. 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 $paths(\mathcal{T})$. The set of finite paths of $\mathcal{T}$, denoted by $finPaths(\mathcal{T})$, is defined as $\{p' \mid p' \sqsubseteq p \text{ for some } p \in paths(\mathcal{T})\}$. Given a path p from a tree $\mathcal{T}$, we define its trace with respect to a labeling function L as $trace_L(p)$. We do not explicitly mention the labeling function if it is clear from the context.

Linear-time and Branching-time Properties. A set of traces $P \subseteq \Sigma^\omega$ is also called a *trace property* or *linear-time property*. As an alternative concept, *branching-time properties* are sets of Σ-labeled trees. The majority of this work is based on linear-time properties but we also study logics for branching-time properties. The class of *ω-regular trace properties* is the class of properties that are recognizable by a Büchi automaton [197].

Definition 2.1. A *Büchi automaton* is a tuple $(Q, q_0, \Sigma, \delta, F)$, where Q is a finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $q_0 \in Q$ the initial state. If δ is a (partial) function, the automaton is deterministic. Furthermore, $F \subseteq Q$ is a set of accepting states. A trace $t \in \Sigma^\omega$ is accepted by a Büchi automaton $\mathcal{A}$ if there exists an infinite run $r \in Q^\omega$ such that r visits states in F infinitely often, $r[0] = q_0$, and for all i, $(r[i], t[i], r[i+1]) \in \delta$.

We allow automata to be partial, even if they are deterministic. To change that, one can add a non-accepting sink state.

The set of safety properties [158] is one of the most successful classes of linear properties as it is amendable to easier monitoring and verification than arbitrary ω-regular properties [154, 133]. A safety property describes a set of finite prefixes which violate the property. Intuitively, they state which “bad things” are not allowed to occur. Their counterpart, liveness properties [158], are properties that describe that that system must always be able to return to a “good” state.

Definition 2.2. A property P is a *safety* property if it holds that for every trace $t \notin P$, there exists a $u \sqsubseteq t$ such that for every t' with $u \sqsubseteq t'$, we have $t' \notin P$. A property P is a *liveness* property if for every $u \in \Sigma^*$, there exists a $t \in \Sigma^\omega$ with $u \sqsubseteq t$ and $t \in P$.

Formulated alternatively, safety properties describe a *safety region*, which should not be left by a trace. This safety region can be defined as safety automata [154].

Definition 2.3. A *safety automaton* is a Büchi automaton in which $F = Q$, i.e., all states are accepting. A trace is thus accepted by a safety automaton iff it has a run through the automaton.

Theorem 2.1 ([154]). *For every ω-regular safety property P, there is a safety automaton that accepts exactly those traces that are in P.*

Hyperproperties. Set-theoretically, hyperproperties are a simple generalization of trace properties. A *hyperproperty* is a set of sets of traces $H \subseteq 2^{(\Sigma^\omega)}$ [51]. While properties are a collection of "accepted" traces, hyperproperties can be seen as a collection of accepted systems, where a system is described by the set of traces it can produce. That way, hyperproperties can relate the traces of a system.

The concept of safety and liveness naturally extends to hyperproperties, resulting in the class of hypersafety and hyperliveness properties. We lift the prefix relation $\sqsubseteq$ to sets of traces: a set of finite traces $U \subseteq \Sigma^*$ is a prefix of a set $T \subseteq \Sigma^\omega$ (written $U \sqsubseteq T$) if, for every $u \in U$, there exists a $t \in T$ such that $u \sqsubseteq t$.

Definition 2.4 ([51])**.** A hyperproperty H is a *hypersafety* property if for every $T \subseteq \Sigma^\omega$ with $T \notin H$, there exists a finite set $U \subseteq \Sigma^*$ with $U \sqsubseteq T$ such that, for every $T' \subseteq \Sigma^\omega$ with $U \sqsubseteq T'$, we have $T' \notin H$. A property H is a *hyperliveness* property if for every finite set $U \subseteq \Sigma^*$, there exists $T \subseteq \Sigma^\omega$ with $U \sqsubseteq T$ and $T \in H$.

Intuitively, a violation of a hypersafety property can be explained by the finite interaction of finitely many traces. Conversely, a hyperproperty is hyperliveness, if such a set can always be extended to a set satisfying the property.

2.2 Models of Computation

We employ multiple models of different expressiveness to describe a system. In the first part of the thesis, we use finite-state transition systems for modeling reactive systems. As Turing-complete models, we use n-counter machines and Turing machines.

Projections and Functions. Given a tuple $(x_1, \ldots, x_n)$, we define the *projection* on the ith component as $\#_i(x) = x_i$. For sets X and Y, we define the projection of X on Y as $X_{|Y} = \{x \mid x \in X \cap Y\}$. If Z is another set, we define $X =_Z Y$ as $X_{|Z} = Y_{|Z}$. Given a function $f : X \to Y$, $x \in X$ and $y \in Y$, we write $f[x \mapsto y]$ for the function that returns y for x and $f(x')$ for $x' \neq x$. An *assignment* of type $X \to Y$ is a (potentially partial) function of type $X \to Y$. We use $\emptyset$ for the empty assignment.

Atomic Propositions. We describe the state of a reactive system with a countable set of *atomic propositions* AP. Thus, in this thesis, Σ is mostly given as 2^{AP}. Given a symbol π, we write AP_π for the set $\{a_\pi \mid a \in AP\}$. For $A \subseteq AP$, we lift the notation $=_A$ to various constructs like functions and traces. For a set X and two labeling functions $L, L' : X \to 2^{AP}$, we write $L =_A L'$ if for every $x \in X$, $L(x) =_A L'(x)$. Similarly, for two traces t, t' over 2^{AP}, we write $t =_A t'$ if for all i, $t[i] =_A t'[i]$. For assignments $\Pi, \Pi' : X \to T$ that map into trace sets, $\Pi =_A \Pi'$ if for all $x \in X$, $\Pi(x) =_A \Pi'(x)$. Lastly, for sets of traces T and T', we use $T =_A T'$ if there exists a total and surjective relation $R \subseteq T \times T'$ such that $t =_A t'$ for all $(t, t') \in R$.

Transition Systems. Transition systems are abstract models of (reactive) systems. Compared with automata, they do not have an acceptance condition. A labeled transition system $\mathcal{S}$ over Σ is a tuple (S, S_0, δ), where S is a set of states, $S_0 \subseteq S$ is a set of initial states and $\delta \subseteq S \times \Sigma \times S$ is a transition relation. In this thesis, we use transition systems with labeled transitions. State-labeled and transition-labeled transition systems can be easily transformed into one another. We assume that every state $s \in S$ has a successor s' such that $(s, A, s') \in \delta$ for some A. In a finite-state system, S and Σ are finite, in infinite-state systems, S and Σ may be infinite. A finite-state transition system over $\Sigma = 2^{AP}$ is also called a *Kripke structure*, although Kripke structures are traditionally given as state-labeled transition systems. We call a transition system deterministic if $|S_0| = 1$ and for each $s \in S$ and $A \in \Sigma$, $|\{(s, A, s') \in \delta\}| \leq 1$. An infinite sequence $t \in \Sigma^\omega$ is a trace of $\mathcal{S}$ if there is an infinite sequence of states $r \in S^\omega$ such that $r[0] \in S_0$ and $(r[i], t[i], r[i+1]) \in \delta$ for all points in time $i \in \mathbb{N}$ with $i \geq 0$. Similarly, a run is finite if $r \in S^+$, resulting in a finite trace $t \in \Sigma^+$. For a run $r \in S^\omega$, we write $\mathit{trace}(r)$ for the (finite or infinite) trace generated by r. We denote the set of all traces of a transition system $\mathcal{S}$ by $\mathit{traces}(\mathcal{S})$ and the set of all finite traces by $\mathit{finTraces}(\mathcal{S})$.

Turing Machines. We use Turing machines in reductions for undecidability results. We define Turing machines as deterministic machines, as every nondeterministic Turing machine can be simulated by a deterministic one. Turing machines are machines that read from and write on a tape of infinite length while keeping track of a state. A *deterministic Turing machine* (TM) is a tuple $(Q, q_0, \Gamma, \delta, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, Γ is a finite alphabet, $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$ is the transition relation and $F \subseteq Q$ is a set of accepting states. A transition $((q, a), (q', a', d)) \in \delta$ means that the if the TM is in state q and reads a, it updates its state to q', writes a', and moves either to the left ($d = L$) or the right ($d = R$). As we define deterministic Turing machines, we require that for each $q \in Q$ and $a \in \Gamma$, $|\{(q', a', d) \mid (q, a), (q', a', d) \in \delta\}| \leq 1$. We assume a dedicated blank symbol $\# \in \Gamma$. In the initial configuration, the head is at position 0, the current state is q_0, and the tape contains some finite word $w \in \Gamma^*$ (not containing #), followed by infinitely many #. We say that a TM accepts an initial word w if there is a run starting in the initial configuration with w on the tape and that eventually visits a configuration with the current state in F.

Counter Machines. Counter machines manipulate n counters with different types of instructions. To achieve Turing-completeness, 2 counters are already sufficient [181]. A deterministic 2-counter machine (2CM) consists of a finite set of m instructions $l_1, \ldots, l_{m-1}, l_m$, where all except the last are of one of the following forms:

- $c_i \coloneqq c_i + 1$; `goto` l_j (for $i \in \{1, 2\}$ and $1 \leq j \leq m$)
- `if` $c_i = 0$ `then goto` l_j `else` $c_i \coloneqq c_i - 1$; `goto` l_k (for $i \in \{1, 2\}$ and $1 \leq j, k \leq m$).

The last instruction is the designated halting instruction, for which we also write l_{halt}. A 2CM configuration s is a triple (i, m, n), which indicates that the values of the two counters are currently m and n and that the next instruction to be executed is l_i. We call each configuration in which i denotes the halting instruction a halting configuration s_{halt}. Furthermore, we say that a 2CM $\mathcal{M}$ halts for a given initial configuration s_0 if there is a finite sequence $s_0, s_1, \ldots, s_{halt}$ such that for all two successive configurations s_i, s_{i+1}, the latter one is a result of applying the instruction specified in s_i to configuration s_i.

The Complexity of Undecidable Problems. Many problems considered in this paper are highly undecidable. To enable precise quantification of "how undecidable", we briefly recall the arithmetic and analytical hierarchies. We only provide a brief overview and refer to [199] for details. The arithmetic hierarchy contains all problems (languages) that can be expressed in first-order arithmetic over the natural numbers. It contains the class of recursively enumerable (RE) and co-enumerable problems (coRE) in its first level. Deciding whether a Turing machine accepts the empty (or any other) initial word is famously RE-complete. Deciding whether a 2CM has a halting computation starting with some initial counter values is equally RE-complete [181]. The class Σ_1^1 (sitting in the analytical hierarchy) contains all problems that can be expressed with existential second-order quantification (over sets of numbers) followed by a first-order arithmetic formula. Analogously, the class Π_1^1 contains all problems expressible using universal second-order quantification. Consequently, both Σ_1^1 and Π_1^1 (strictly) contain the entire arithmetic hierarchy.

2.3 Definitions of Logics

In this section, we define a variety of logics on which we build in this thesis, especially in →Chapter 3. We first define linear-time logics, which are evaluated with respect to traces $t \in (2^{AP})^\omega$. Linear-time logics thus define linear-time properties. We then discuss the relevant branching-time logics, which are evaluated with respect to AP-labeled trees $(\mathcal{T}, L)$. Finally, we describe linear-time hyperlogics with respect to sets of traces $T \subseteq (2^{AP})^\omega$, and branching-time hyperlogics, which are also evaluated on AP-labeled trees. →page 32

For all of these logics, we define their satisfaction relation $M \models \varphi$, where $M \in \{t, T, (\mathcal{T}, L)\}$ is the corresponding type of model and φ is a formula of the logic. We say that a formula φ is *satisfiable* if there exists an M such that $M \models \varphi$. If M is a set of traces, we additionally require that M is non-empty. Given two logics $\mathcal{L}_1, \mathcal{L}_2$ over the same type of model, we say that formulas $\varphi_1 \in \mathcal{L}_1$ and $\varphi_2 \in \mathcal{L}_2$ are *equivalent* (written $\varphi_1 \equiv \varphi_2$) if for all instances M of the type of model, $M \models \varphi_1$ iff $M \models \varphi_2$. We say that $\mathcal{L}_2$

is *at least as expressive* as $\mathcal{L}_1$ if for every $\mathcal{L}_1$-formula φ_1, there exists an $\mathcal{L}_2$-formula φ_2 such that $\varphi_1 \equiv \varphi_2$. In this case, we also say that $\mathcal{L}_2$ *subsumes* $\mathcal{L}_1$. They are *expressively equivalent*, if $\mathcal{L}_1$ is at least as expressive as $\mathcal{L}_2$ and vice versa.

2.3.1 Linear-time Logics

Linear-time logics describe trace properties. All linear-time logics we define in this part of the thesis describe ω-regular trace properties.

LTL. Linear temporal logic (LTL) [190] combines Boolean connectives with temporal modalities $\bigcirc$ (next) and $\mathcal{U}$ (until). The syntax is given by the following grammar:

$$\psi ::= a \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc\psi \mid \psi \,\mathcal{U}\, \psi$$

where $a \in AP$. The Boolean connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ can be derived from the syntax, as well as the LTL modalities eventually $\Diamond\psi \equiv \mathit{true}\,\mathcal{U}\,\psi$, globally $\Box\psi \equiv \neg\Diamond\neg\psi$, and weak until $\psi_1 \,\mathcal{W}\, \psi_2 \equiv (\psi_1 \,\mathcal{U}\, \psi_2) \vee \Box\psi_1$. Given a trace $t \in (2^{AP})^\omega$ and a point in time $i \in \mathbb{N}$, the semantics of an LTL formula is defined as follows:

$$\begin{array}{lll}
t, i \models a & \text{iff} & a \in t[i] \\
t, i \models \neg\psi & \text{iff} & t, i \not\models \psi \\
t, i \models \psi_1 \vee \psi_2 & \text{iff} & t, i \models \psi_1 \text{ or } t, i \models \psi_2 \\
t, i \models \bigcirc\psi & \text{iff} & t, i+1 \models \psi \\
t, i \models \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \exists j \geq i.\, t, j \models \psi_2 \text{ and } \forall i \leq k < j.\, t, k \models \psi_1
\end{array}$$

A trace t satisfies an LTL formula ψ, written $t \models \psi$, if $t, 0 \models \psi$.

QPTL. QPTL [209] extends LTL with quantification over atomic propositions. QPTL formulas ψ are defined as follows. It can express all ω-regular trace properties [146].

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

where $a \in AP$. We did not define the until operator $\mathcal{U}$ as native part of QPTL as it can be derived using propositional quantification [143]. Additional operators like $\Box$ can be derived as in LTL. For simplicity, we assume that propositional variable names in formulas are cleared of double occurrences. QPTL inherits the semantics of LTL with additional rules for the propositional quantification.

$$t, i \models \exists a.\, \psi \quad \text{iff} \quad \exists t' \in (2^{AP})^\omega.\, t' =_{AP \setminus \{a\}} t \text{ and } t', i \models \psi$$

$$t, i \models \forall a.\, \psi \quad \text{iff} \quad \forall t' \in (2^{AP})^{\omega}.\ \text{if } t' =_{AP\setminus\{a\}} t \text{ then } t', i \models \psi$$

FO[<]. First-order monadic logic of order, written FO[<], is a first-order logic evaluated on traces. Let $V_1 = \{x, y, \ldots\}$ be a set of first-order variables. The syntax of FO[<] consists of first-order quantification, unary predicates P_a for every $a \in AP$, and two binary relations $<$ and $=$.

$$\psi ::= P_a(x) \mid x < y \mid x = y \mid \neg\psi \mid \psi \vee \psi \mid \exists x.\, \psi$$

As for LTL, we evaluate FO[<] with respect to a trace t over AP. Additionally, let $\mathcal{V}_1 : V_1 \to \mathbb{N}$ be an assignment function for the variables. The interpretation of $<$ and $=$ is fixed as the respective interpretation on $\mathbb{N}$. We define the semantics of FO[<] formulas as follows.

$$\begin{array}{lll}
\mathcal{V}_1, t \models P_a(x) & \text{iff} & a \in t[\mathcal{V}_1(x)] \\
\mathcal{V}_1, t \models x < y & \text{iff} & \mathcal{V}_1(x) < \mathcal{V}_1(y) \\
\mathcal{V}_1, t \models x = y & \text{iff} & \mathcal{V}_1(x) = \mathcal{V}_1(y) \\
\mathcal{V}_1, t \models \neg\psi & \text{iff} & \mathcal{V}_1, t \not\models \psi \\
\mathcal{V}_1, t \models \psi_1 \vee \psi_2 & \text{iff} & \mathcal{V}_1, t \models \psi_1 \text{ or } \mathcal{V}_1, t \models \psi_2 \\
\mathcal{V}_1, t \models \exists x.\, \psi & \text{iff} & \exists i \in \mathbb{N}.\, \mathcal{V}_1[x \mapsto i], t \models \psi
\end{array}$$

A FO[<] formula is closed if it does not contain free occurrences of variables. A trace over AP satisfies a closed formula ψ, written $t \models \psi$, iff $\emptyset, t \models \psi$. A FO[<] formula ψ is in *prenex normal form* if ψ starts with a sequence of quantifiers followed by a quantifier-free formula. Every FO[<] formula can be transformed into an equivalent formula in prenex normal form. The same holds for all FO/SO logics considered in this work.

Kamp's theorem [144] (in the formulation of [109]) establishes that LTL and FO[<] are expressively equivalent. This might be surprising considering that LTL is a modal logic whereas FO[<] has the full power of first-order quantification.

Theorem 2.2 (Kamp's Theorem[144, 109])**.** *LTL and FO[<] are equally expressive.*

S1S. S1S, the second-order logic of one successor [36], is strictly more expressive than FO[<] due to its second-order quantifiers. Let V_1 be a set of first-order variables as before and let $V_2 = \{X, Y, \ldots\}$ be a set of second-order variables. Compared with FO[<], the terms of S1S do not contain the $<$ relation but a function S for the successor of a term and the constant *min* to address the minimal element.

$$\begin{array}{l}
\tau ::= x \mid min \mid S(\tau) \\
\psi ::= \tau \in X \mid \tau = \tau \mid \neg\psi \mid \psi \vee \psi \mid \exists x.\, \psi \mid \exists X.\, \psi
\end{array}$$

Here, $X \in V_2 \cup \{X_a \mid a \in AP\}$. We interpret S1S formulas with respect to a first-order valuation $\mathcal{V}_1 : V_1 \to \mathbb{N}$ and a second-order valuation $\mathcal{V}_2 : V_2 \to 2^{\mathbb{N}}$. The value of a term is defined as:

$$\begin{aligned} [x]_{\mathcal{V}_1} &= \mathcal{V}_1(x) \\ [min]_{\mathcal{V}_1} &= 0 \\ [S(\tau)]_{\mathcal{V}_1} &= [\tau]_{\mathcal{V}_1} + 1 \end{aligned}$$

Let ψ be an S1S formula with free first-order variables $V_1' \subseteq V_1$ and free second-order variables $V_2' \subseteq V_2 \cup \{X_a \mid a \in AP\}$. We define the satisfaction relation $\mathcal{V}_1, \mathcal{V}_2 \models \psi$ with respect to two valuations $\mathcal{V}_1, \mathcal{V}_2$ assigning all free variables in V_1' and V_2' as follows.

$$\begin{aligned} \mathcal{V}_1, \mathcal{V}_2 &\models \tau \in X && \text{iff} && [\tau]_{\mathcal{V}_1} \in \mathcal{V}_2(X) \\ \mathcal{V}_1, \mathcal{V}_2 &\models \tau_1 = \tau_2 && \text{iff} && [\tau_1]_{\mathcal{V}_1} = [\tau_2]_{\mathcal{V}_1} \\ \mathcal{V}_1, \mathcal{V}_2 &\models \neg\psi && \text{iff} && \mathcal{V}_1, \mathcal{V}_2 \not\models \psi \\ \mathcal{V}_1, \mathcal{V}_2 &\models \psi_1 \vee \psi_2 && \text{iff} && \mathcal{V}_1, \mathcal{V}_2 \models \psi_1 \text{ or } \mathcal{V}_1, \mathcal{V}_2, t \models \psi_2 \\ \mathcal{V}_1, \mathcal{V}_2 &\models \exists x.\, \psi && \text{iff} && \exists i \in \mathbb{N}.\, \mathcal{V}_1[x \mapsto i], \mathcal{V}_2 \models \psi \\ \mathcal{V}_1, \mathcal{V}_2 &\models \exists X.\, \psi && \text{iff} && \exists A \subseteq \mathbb{N}.\, \mathcal{V}_1, \mathcal{V}_2[X \mapsto A] \models \psi \end{aligned}$$

We call an S1S formula ψ closed if every free variable is a second-order variable of the form X_a with $a \in AP$. We say that a trace t over 2^{AP} satisfies a closed S1S formula ψ, written $t \models \psi$, if $\emptyset, \mathcal{V}_2 \models \psi$, where $\emptyset$ denotes the empty first-order valuation and $\mathcal{V}_2$ is the valuation that assigns each free X_a in ψ to the set $\{i \in \mathbb{N} \mid a \in t[i]\}$.

In a result similar to Kamp's theorem, it was shown that QPTL and S1S are equally expressive. This result in particular implies that both logics have the same expressiveness as Büchi automata.

Theorem 2.3 ([146]). *QPTL and S1S are equally expressive.*

2.3.2 Branching-time Logics

CTL*. CTL* is a branching-time logic that extends the syntax LTL with a path quantifier E that existentially quantifies a branch from a node in a tree. The syntax, where ψ denotes state formulas and θ denotes path formulas, is given as follows:

$$\begin{aligned} \psi &::= a \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{E}\,\theta \\ \theta &::= \psi \mid \neg\theta \mid \theta \vee \theta \mid \mathsf{X}\,\theta \mid \theta\, \mathsf{U}\, \theta \end{aligned}$$

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

$$\begin{array}{lll}
s \models_{(\mathcal{T},L)} a & \text{iff} & a \in L(s) \\
s \models_{(\mathcal{T},L)} \neg\psi & \text{iff} & s \not\models_{(\mathcal{T},L)} \psi \\
s \models_{(\mathcal{T},L)} \psi_1 \vee \psi_2 & \text{iff} & s \models_{(\mathcal{T},L)} \psi_1 \text{ or } s \models_{(\mathcal{T},L)} \psi_2 \\
s \models_{(\mathcal{T},L)} \mathsf{E}\,\theta & \text{iff} & \exists p \in \mathit{paths}(\mathcal{T}, s).\, p \models_{(\mathcal{T},L)} \theta \\
p \models_{(\mathcal{T},L)} \psi & \text{iff} & p[0] \models_{(\mathcal{T},L)} \psi \\
p \models_{(\mathcal{T},L)} \neg\theta & \text{iff} & p \not\models_{(\mathcal{T},L)} \theta \\
p \models_{(\mathcal{T},L)} \theta_1 \vee \theta_2 & \text{iff} & p \models_{(\mathcal{T},L)} \theta_1 \text{ or } p \models_{(\mathcal{T},L)} \theta_2 \\
p \models_{(\mathcal{T},L)} \mathsf{X}\,\theta & \text{iff} & p[1,\infty] \models_{(\mathcal{T},L)} \theta \\
p \models_{(\mathcal{T},L)} \theta_1 \,\mathsf{U}\, \theta_2 & \text{iff} & \exists i \geq 0.\, p[i,\infty] \models_{(\mathcal{T},L)} \theta_2 \text{ and } \forall 0 \leq j < i.\, p[j,\infty] \models_{(\mathcal{T},L)} \theta_1
\end{array}$$

Similar to LTL, we use $\mathsf{F}\,\theta \equiv \mathit{true}\,\mathsf{U}\,\theta$ and $\mathsf{G}\,\theta \equiv \neg\,\mathsf{F}\,\neg\theta$. For a labeled tree $(\mathcal{T}, L)$ and a CTL* formula ψ, we write $(\mathcal{T}, L) \models \psi$ if $\mathcal{T}$ has root r such that $r \models_{(\mathcal{T},L)} \psi$.

QCTL*. Similarly to how QPTL extends LTL with the possibility to reassign atomic propositions, CTL* can be extended to QCTL* [108]. In [108], QCTL* has been introduced with different semantics: two which evaluate formulas direclty on finite transition systems and one which evaluates the formulas on trees. As we evaluate temporal logics with respect to traces and trees and not with respect to the underlying transition system, we follow the tree semantics of the logic. We extend the syntax of CTL* state formulas with propositional quantification.

$$\psi ::= a \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{E}\,\theta \mid \exists a.\,\psi$$

The semantics for the additional rule is given as follows.

$$s \models_{(\mathcal{T},L)} \exists a.\,\psi \quad \text{iff} \quad \exists L'.\, L' =_{AP\setminus\{a\}} L \text{ and } s \models_{(\mathcal{T},L')} \psi$$

MPL. Monadic path logic (MPL) [4] is a second-order branching-time logic. MPL is syntactically similar to FO[<] but has additional second-order variables and quantifiers. As before, let $V_1 = \{x, y, \ldots\}$ be a set first-order variables and $V_2 = \{X, Y, \ldots\}$ be a set of second-order variables. We define the set of MPL formulas as follows.

$$\psi ::= x \in X \mid x < y \mid x = y \mid \neg\psi \mid \psi \vee \psi \mid \exists x.\,\psi \mid \exists X.\,\psi$$

As a branching-time logic, MPL is interpreted over AP-labeled trees $(\mathcal{T}, L)$. First-order variables are assigned nodes in the tree and second-order variables sets of nodes. We represent a node by the finite path leading to it. Let $\mathcal{V}_1 : V_1 \to \mathit{finPaths}(\mathcal{T})$ and

$\mathcal{V}_2 : V_2 \rightarrow 2^{\mathit{finPaths}(\mathcal{T})}$. Let ψ be a MPL formula with free first-order variables $V_1' \subseteq V_1$ and free second-order variables $V_2' \subseteq V_2 \cup \{X_a \mid a \in AP\}$. We define the satisfaction relation $\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} \psi$ as follows, where $\mathcal{V}_1$ and $\mathcal{V}_2$ assign all free variables V_1' and V_2'.

$$\begin{array}{lll}
\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} x \in X & \text{iff} & \mathcal{V}_1(x) \in \mathcal{V}_2(X) \\
\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} x < y & \text{iff} & \mathcal{V}_1(x) \sqsubseteq \mathcal{V}_1(y) \\
\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} x = y & \text{iff} & \mathcal{V}_1(x) = \mathcal{V}_1(y) \\
\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} \neg\psi & \text{iff} & \mathcal{V}_1, \mathcal{V}_2 \not\models_{(\mathcal{T},L)} \psi \\
\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} \psi_1 \vee \psi_2 & \text{iff} & \mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} \psi_1 \text{ or } \mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} \psi_2 \\
\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} \exists x.\, \psi & \text{iff} & \exists p \in \mathit{finPaths}(\mathcal{T}).\, \mathcal{V}_1[x \mapsto p], \mathcal{V}_2 \models_{(\mathcal{T},L)} \psi \\
\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} \exists X.\, \psi & \text{iff} & \exists p \in \mathit{paths}(\mathcal{T}). \\
 & & \qquad \mathcal{V}_1, \mathcal{V}_2[X \mapsto \{p' \in \mathit{finPaths}(\mathcal{T}) \mid p' \sqsubseteq p\}] \models_{(\mathcal{T},L)} \psi
\end{array}$$

MPL's second-order quantification can only quantify full paths of $\mathcal{T}$. A second-order variables X is therefore assigned with the set of all prefixes of a path p of $\mathcal{T}$. We call an MPL formula ψ closed if every free variable is a second-order variable of the form X_a with $a \in AP$. We say that an AP-labeled tree $(\mathcal{T}, L)$ satisfies a closed MPL formula ψ, written $(\mathcal{T}, L) \models \psi$, if $\emptyset, \mathcal{V}_2 \models \psi$, where $\mathcal{V}_2$ is the valuation that assigns each free X_a in ψ to the set $\{p \in \mathit{finPaths}(\mathcal{T}) \mid |p| = n \text{ and } a \in L(p[n-1]) \text{ for some } n \in \mathbb{N}_{>0}\}$.

Theorem 2.4 ([182]). *MPL and CTL* are expressively equivalent.*

MSO. Monadic second-order logic (MSO) interpreted over labeled trees shares the syntax and most of its semantics with MPL. The only difference is the valuation of the second-order quantification, where we lift the requirement that the quantified set encodes a full path.

$$\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} \exists X.\, \psi \quad \text{iff} \quad \exists P \subseteq \mathit{finPaths}(\mathcal{T}).\, \mathcal{V}_1, \mathcal{V}_2[X \mapsto P] \models_{(\mathcal{T},L)} \psi$$

Theorem 2.5 ([162]). *MSO and QCTL* are expressively equivalent.*

2.3.3 Linear-time Hyperlogics

HyperLTL. HyperLTL [50] extends LTL with explicit trace quantification. Let $V_\pi = \{\pi_1, \pi_2, \ldots\}$ be an infinite set of trace variables. HyperLTL formulas are defined by the grammar:

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

where $a \in AP$ and $\pi \in V_\pi$. Here, $\forall \pi.\, \varphi$ and $\exists \pi.\, \varphi$ denote universal and existential trace quantification, and a_π requires the atomic proposition a to hold on trace π. The semantics of HyperLTL is defined with respect to a set of traces T. Let $\Pi : V_\pi \rightarrow T$ be a trace assignment that maps trace variables to traces in T. The satisfaction relation for HyperLTL is defined with respect a set of traces T.

$$\begin{array}{lll}
\Pi, i \models_T a_\pi & \text{iff} & a \in \Pi(\pi)[i] \\
\Pi, i \models_T \neg\varphi & \text{iff} & \Pi, i \not\models_T \varphi \\
\Pi, i \models_T \varphi_1 \vee \varphi_2 & \text{iff} & \Pi, i \models_T \varphi_1 \text{ or } \Pi, i \models_T \varphi_2 \\
\Pi, i \models_T \bigcirc \varphi & \text{iff} & \Pi, i+1 \models_T \varphi \\
\Pi, i \models_T \varphi_1 \mathcal{U} \varphi_2 & \text{iff} & \exists j \geq i.\ \Pi, j \models_T \varphi_2 \text{ and } \forall i \leq k < j.\ \Pi, k \models_T \varphi_1 \\
\Pi, i \models_T \exists \pi.\, \varphi & \text{iff} & \exists t \in T.\, \Pi[\pi \mapsto t], i \models_T \varphi \\
\Pi, i \models_T \forall \pi.\, \varphi & \text{iff} & \forall t \in T.\, \Pi[\pi \mapsto t], i \models_T \varphi
\end{array}$$

A trace set T satisfies a HyperLTL formula φ, written $T \models \varphi$, if $\emptyset, 0 \models_T \varphi$. HyperLTL formulas are syntactically required to be in negation normal formal. Nevertheless, they are closed under Boolean connectives [50].

To refer to HyperLTL formula, we often use $Q\pi.\, \psi$ to denote a formula starting with either an existential or a universal quantifier. We also use regular expressions to define quantifier prefixes, e.g., $Q^n\forall^*$ denotes the set of HyperLTL formulas that start with n arbitrary quantifiers followed by an arbitrary number of $\forall$ quantifiers. This also applies to any extension of HyperLTL.

HyperQPTL. HyperQPTL [195] extends QPTL with explicit trace quantification. We add atomic formulas p, which are independent of the trace variables, and prenex propositional quantification $\exists p.\, \varphi$ to the syntax.

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

HyperQPTL inherits the semantics of HyperLTL with additional rules for the new syntactic constructs.

$$\begin{array}{lll}
\Pi, i \models_T p & \text{iff} & \forall t \in T.\, p \in t[i] \\
\Pi, i \models_T \exists p.\, \varphi & \text{iff} & \exists t_p \in (2^{\{p\}})^\omega, T' \subseteq (2^{AP})^\omega, \Pi' : V_\pi \rightarrow T'.\, \forall t \in T'.\, t =_{\{p\}} t_p \text{ and} \\
 & & \quad T =_{AP\setminus\{p\}} T' \text{ and } \Pi =_{AP\setminus\{p\}} \Pi' \text{ and } \Pi', i \models_{T'} \varphi \\
\Pi, i \models_T \forall p.\, \varphi & \text{iff} & \forall t_p \in (2^{\{p\}})^\omega, T' \subseteq (2^{AP})^\omega, \Pi' : V_\pi \rightarrow T'.\, \text{if } \forall t \in T'.\, t =_{\{p\}} t_p \text{ and} \\
 & & \quad T =_{AP\setminus\{p\}} T' \text{ and } \Pi =_{AP\setminus\{p\}} \Pi', \text{ then } \Pi', i \models_{T'} \varphi
\end{array}$$

LTL with Team Semantics. Team semantics are an alternative semantics to the classic Tarski semantics. In the case of LTL, team semantics maintain the syntax of the logic but assign a semantics based on sets traces. TeamLTL thus constitutes a linear-time hyperlogic. As a hyperlogic, TeamLTL was first studied in [153], where it was called LTL with synchronous team semantics. Logics with team semantics are typically not closed under classical negation and adding Boolean negation usually adds a considerable amount of expressiveness. We therefore define the syntax of TeamLTL in *negation normal form*, where negation only occurs directly before atomic propositions.

$$\psi ::= a \mid \neg a \mid \psi \wedge \psi \mid \psi \vee \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi \mid \psi \,\mathcal{W}\, \psi$$

We define the satisfaction relation $T, i \models \varphi$ of TeamLTL as follows.

$$\begin{array}{lll}
T, i \models a & \text{iff} & \forall t \in T.\, a \in t[i] \\
T, i \models \neg a & \text{iff} & \forall t \in T.\, a \notin t[i] \\
T, i \models \psi_1 \wedge \psi_2 & \text{iff} & T, i \models \psi_1 \text{ and } T, i \models \psi_2 \\
T, i \models \psi_1 \vee \psi_2 & \text{iff} & \exists T_1, T_2.\, T_1 \cup T_2 = T \text{ and } T_1, i \models \psi_1 \text{ and } T_2, i \models \psi_2 \\
T, i \models \bigcirc \psi & \text{iff} & T, i+1 \models \psi \\
T, i \models \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \exists j \geq i.\, T, j \models \psi_2 \text{ and } \forall i \leq k < j.\, T, k \models \psi_1 \\
T, i \models \psi_1 \,\mathcal{W}\, \psi_2 & \text{iff} & \forall j \geq i.\, T, j \models \psi_1 \text{ or } \exists j \leq k.\, T, k \models \psi_2
\end{array}$$

The interesting part of the definition is the evaluation of the $\vee$ operator. It splits the set and requires that each subsets satisfies one of the subformulas. In particular, one of the trace sets might also be empty. Note that $T, i \models \mathit{false}$ iff $T = \emptyset$. The pair (T, i) is called a *team*.

FO[$<, E$]. The linear hyperlogic FO[$<, E$] [95] extends FO[$<$] with a binary predicate E, called *equal-level predicate*, which relates points in time. The syntax of FO[$<, E$] is obtained by extending the syntax of FO[$<$] with $E(x, y)$. Given a set V_1 of first-order variables, we define the syntax of FO[$<, E$] formulas as follows.

$$\begin{aligned}
\tau &::= P_a(x) \mid x < y \mid x = y \mid E(x, y) \\
\varphi &::= \tau \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x.\, \varphi
\end{aligned}$$

where $a \in AP$ and $x, y \in V_1$. While FO[$<$] formulas are interpreted over a trace t, we interpret an FO[$<, E$] formula φ over a set of traces T. Compared with FO[$<$], we assign first-order variables with elements from the domain $T \times \mathbb{N}$. Let φ be an FO[$<, E$] formula with free first-order variables $V_1' \subseteq V_1$. We define the satisfaction relation $\mathcal{V}_1 \models_T \varphi$ with respect to T and a first-order valuation $\mathcal{V}_1 : V_1 \rightarrow T \times \mathbb{N}$, which assigns all variables free variables V_1' of φ.

$$\begin{array}{lll}
\mathcal{V}_1 \models_T P_a(x) & \text{iff} & a \in t[i] \text{ for } t = \#_1(\mathcal{V}_1(x)) \text{ and } i = \#_2(\mathcal{V}_1(x)) \\
\mathcal{V}_1 \models_T x < y & \text{iff} & \#_1(\mathcal{V}_1(x)) = \#_1(\mathcal{V}_1(y)) \text{ and } \#_2(\mathcal{V}_1(x)) < \#_2(\mathcal{V}_1(y)) \\
\mathcal{V}_1 \models_T x = y & \text{iff} & \mathcal{V}_1(x) = \mathcal{V}_1(y) \\
\mathcal{V}_1 \models_T E(x,y) & \text{iff} & \#_2(\mathcal{V}_1(x)) = \#_2(\mathcal{V}_1(y)) \\
\mathcal{V}_1 \models_T \neg\psi & \text{iff} & \mathcal{V}_1 \not\models_T \psi \\
\mathcal{V}_1 \models_T \psi_1 \vee \psi_2 & \text{iff} & \mathcal{V}_1 \models_T \psi_1 \text{ or } \mathcal{V}_1 \models_T \psi_2 \\
\mathcal{V}_1 \models_T \exists x.\,\psi & \text{iff} & \exists t \in T.\, \exists i \in \mathbb{N}.\, \mathcal{V}_1[x \mapsto (t,i)] \models_T \psi
\end{array}$$

Recall that $\#_i(x)$ defines the ith →projection on x. We use the $<$ predicate to enforce that two points lie on the same trace. A trace set T satisfies a closed FO[$<, E$] formula φ, written $T \models \varphi$, if $\emptyset \models_T \varphi$. As opposed to LTL and FO[$<$], HyperLTL and FO[$<, E$] are *not* equally expressive. →page 20

Theorem 2.6 ([95]). *FO[$<, E$] is strictly more expressive than HyperLTL.*

2.3.4 Branching-time Hyperlogics

Branching-time hyperlogics are, as standard branching-time logics, interpreted on labeled trees. The increased expressiveness of branching-time hyperlogics lies in the fact that hyperlogics can relate multiple branches of arbitrary distance.

HyperCTL*. HyperCTL* [50] extends CTL* with explicit path variables and quantification. Quantification in HyperCTL* ranges over the paths in a tree. Let V_π be a set of path variables. HyperCTL* formulas are generated by the following grammar:

$$\varphi ::= a_\pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi \,\mathsf{U}\, \varphi \mid \exists \pi.\, \varphi$$

where $a \in AP$ and $\pi \in V_\pi$. The semantics of a HyperCTL* formula is defined with respect to an AP-labeled tree $(\mathcal{T}, L)$ and a path assignment $\Pi : V_\pi \to paths(\mathcal{T})$.

$$\begin{array}{lll}
\Pi, i \models_{(\mathcal{T},L)} a_\pi & \text{iff} & a \in L(\Pi(\pi)[i]) \\
\Pi, i \models_{(\mathcal{T},L)} \neg\varphi & \text{iff} & \Pi, i \not\models_{(\mathcal{T},L)} \varphi \\
\Pi, i \models_{(\mathcal{T},L)} \varphi_1 \vee \varphi_2 & \text{iff} & \Pi, i \models_{(\mathcal{T},L)} \varphi_1 \text{ or } \Pi, i \models_{(\mathcal{T},L)} \varphi_2 \\
\Pi, i \models_{(\mathcal{T},L)} \mathsf{X}\,\varphi & \text{iff} & \Pi, i+1 \models_{(\mathcal{T},L)} \varphi \\
\Pi, i \models_{(\mathcal{T},L)} \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists j \geq i.\, \Pi, j \models_{(\mathcal{T},L)} \varphi_2 \text{ and } \forall i \leq k < j.\, \Pi, k \models_{(\mathcal{T},L)} \varphi_1 \\
\Pi, i \models_{(\mathcal{T},L)} \exists \pi.\, \varphi & \text{iff} & \exists p \in paths(\mathcal{T}).\, p[0,i] = \Pi(\varepsilon)[0,i] \\
& & \quad \text{and } \Pi[\pi \mapsto p, \varepsilon \mapsto p], i \models_{(\mathcal{T},L)} \varphi
\end{array}$$

where we use ε to denote the last path that was added to the path assignment Π. We say that a AP-labeled tree $(\mathcal{T}, L)$ satisfies a HyperCTL* formula φ, written as $(\mathcal{T}, L) \models \varphi$, if $\emptyset, 0 \models_{(\mathcal{T},L)} \varphi$.

Chapter 3

An Expressiveness Hierarchy of Hyperlogics

In this chapter, we conduct a comprehensive expressiveness study of different types of logics for the specification of hyperproperties. We consider three classes of hyperlogics: temporal hyperlogics, first-order and second-order hyperlogics, and variants of LTL with team semantics.

The first part of the study is inspired by Kamp's famous theorem [144], which states (in the formulation of Gabbay et al. [109]) that LTL and monadic first-order logic of order FO[$<$] are expressively equivalent. This result might seem quite surprising: LTL is a purely modal logic, while FO[$<$] uses first-order quantification to reason about the trace. Both logics have been lifted to hyperlogics: LTL has been extended with trace quantification resulting in HyperLTL, and FO[$<$] has been equipped with a so-called equal-level predicate, resulting in FO[$<, E$]. Contrary to Kamp's theorem, however, the potential analog that FO[$<, E$] and HyperLTL might be expressively equivalent is known *not* to be true [95]. Motivated by this unexpected result, we systematically compare the expressiveness of temporal hyperlogics with trace and path quantifiers and first-order/second-order hyperlogics with the equal-level predicate. We follow the known hierarchy of standard logics. Similar to Kamp's thereom, there exist number of equivalence results for pairs of temporal and FO/SO logics: QPTL and S1S are expressively equivalent, as are the branching-time logics CTL* and MPL as well as QCTL* and MSO.

The results of our study are depicted in Figure 3.1. They show that it is in general the case that the equal-level predicate adds more expressive power to first-order and second-order logics than trace and path quantification adds to an equivalent temporal logic. There are subtle differences, however, which depend on the way we interpret the "straight-forward" extension of a logic to a hyperlogic. Propositional quantifiers in QPTL, for example, reassign an atomic proposition on the trace. Consequently, HyperQPTL, as first defined in [195], reassigns atomic propositions uniformly across

$$\begin{tikzcd}
\text{S1S} = \text{QPTL [146]} \arrow[d, dash, "\vee" description] \\
\text{FO}[<] = \text{LTL [109]}
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\text{MSO} = \text{QCTL}^* \text{ [162]} \arrow[d, dash, "\vee" description] \\
\text{MPL} = \text{CTL}^* \text{ [182]}
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
\text{S1S}[E] = \text{HyperQPTL}^{+} \text{ (Thm. 3.5)} \arrow[d, dash, "\vee \text{ (Thm. 3.7)}"] \\
\text{HyperQPTL} \arrow[d, dash, "\vee \text{ (Thm. 3.2)}"] \\
\text{FO}[<, E] \arrow[d, dash, "\vee \text{ [95]}"] \\
\text{HyperLTL}
\end{tikzcd}$$

(c)

$$\begin{tikzcd}
\text{MSO}[E] = \text{HyperQCTL}^* \text{ (Thm. 3.20)} \arrow[d, dash, "\vee \text{ (Thm. 3.19)}"] \\
\text{HyperQ}^{-}\text{CTL}^* \arrow[d, dash, "\vee \text{ (Thm. 3.18)}"] \\
\text{MPL}[E] \arrow[d, dash, "\vee \text{ (Thm. 3.16)}"] \\
\text{HyperCTL}^*
\end{tikzcd}$$

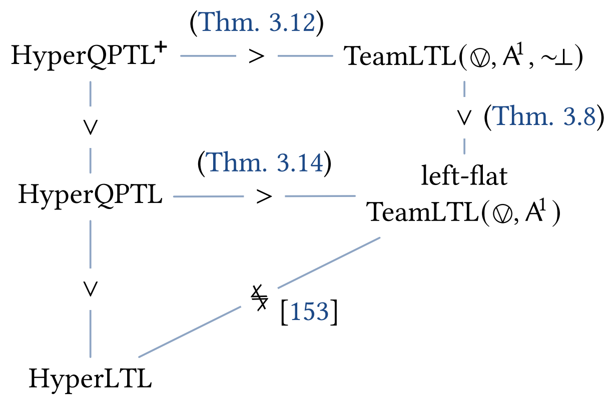
(d)

Figure 3.1: The linear-time hierarchies of standard logics (a) and hyperlogics (c), and the branching-time hierarchies of standard logics (b) and hyperlogics (d). Novel results are annotated with the corresponding theorem number. Commonly known results are not annotated.

all traces. If the quantifier reassigned the proposition differently on every trace, we would obtain a more expressive logic (which we call HyperQPTL^{+}). A similar observation can be made for the branching-time logics QCTL* and HyperQCTL*.

Both temporal and FO/SO hyperlogics rely on some sort of explicit quantification, either trace quantification as in HyperLTL or simultaneous trace and time quantification as in FO$[<, E]$. The algorithmic success of LTL over first-order logics for the specification of linear-time properties stems from the fact that its modal operators replace explicit quantification of points in time. As a consequence, LTL often allows for a more concise and readable formulation of the same property. The natural question to ask is whether a purely modal logic for hyperproperties would have similar advantages. A candidate for such a logic is LTL with team semantics [153]. Under team semantics, LTL expresses hyperproperties without explicit references to traces. Instead, each subformula is evaluated with respect to a set of traces, called a team. Temporal operators advance time on all traces of the current team. The crucial operator in TeamLTL is $\vee$, called *split operator* under team semantics, which splits a set of traces in two.

The second part of our study integrates TeamLTL and extensions thereof into the hierarchy of linear-time hyperlogics. We build on the fact that TeamLTL and HyperLTL are of incomparable expressiveness [153], which already suggests that the logics constitute a fundamentally different approach to express hyperproperties. Indeed, relating the expressiveness of those two types of logics is a challenging task, even if TeamLTL and quantifier-based temporal hyperlogics are both based on temporal logics. The results of the second part of our expressiveness study are illustrated in Figure 3.2. We



(Thm. 3.12)
HyperQPTL$^+$
>
TeamLTL($\varovee$, A^1, $\sim\bot$)
∨
∨ (Thm. 3.8)
(Thm. 3.14)
HyperQPTL
>
left-flat TeamLTL($\varovee$, A^1)
∨
≯ ≮ [153]
HyperLTL

Figure 3.2: Variants of TeamLTL versus linear-time temporal hyperlogics. Novel results are annotated with the corresponding theorem number. Results from earlier sections are not annotated.

focus on two extensions of TeamLTL, TeamLTL($\varovee$, A^1, $\sim\bot$) and TeamLTL($\varovee$, A^1), which can express all (downward-closed) LTL-definable Boolean relations. We show that the expressiveness of TeamLTL($\varovee$, A^1, $\sim\bot$) is captured by HyperQPTL$^+$, which sits at the top of the linear-time hierarchy. For TeamLTL($\varovee$, A^1), we can define a fragment we call *left-flat*, for which we can show inclusion in HyperQPTL.

Outline. Section 3.1 presents the study of temporal and FO/SO hyperlogics for linear-time logics. In Section 3.2, we extend this hierarchy with our results on the expressiveness of variants of TeamLTL. Section 3.3 covers our results on branching-time temporal and FO/SO hyperlogics.

Publications. Section 3.1 is mostly based on [55]. The proposal of HyperQPTL$^+$ as alternative definition of HyperQPTL and the example from Section 3.1.3 stem from [84]. The inclusion of HyperQPTL$^+$ in the linear-time hierarchy was added for this thesis. Section 3.2 contains material from [225]. Section 3.3 is again based on [55]. The definition of HyperQ$^-$CTL* as weaker version of HyperQCTL* was added for this thesis.

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

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

[225] Jonni Virtema, Jana Hofmann, Bernd Finkbeiner, Juha Kontinen, and Fan Yang. **Linear-time Temporal Logic with Team Semantics: Expressivity and Complexity**. *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2021).*

3.1 The Hierarchy of Linear-time Hyperlogics

We commence our study of the expressiveness of hyperlogics with the linear-time hierarchy. Based on the known hierarchy of linear-time trace properties depicted in Figure 3.1a, we compare the expressiveness of hyperlogics that result from extending temporal logics with trace quantification and first-order/second-order logics with the equal-level predicate.

For this section and the rest of the chapter, we require, without loss of generality, that all variables occurring in a formula are distinct. Furthermore, formulas all range over the same set of propositions AP. Where applicable, formulas are also assumed to be closed. In the following, we define a range of translations between different logics. Most of these translations are linear and follow closely the inductive structure of the formula. Once the translation is spelled out, its correctness follows with straightforward inductive reasoning only using the semantics of the respective logics. In such cases, we omit the details of the inductive proofs.

In our translations, we repeatedly use the following HyperLTL formulas, which can also be used in all syntactic extensions of HyperLTL. Formula $once(a, \pi)$ indicates that proposition a holds exactly once on trace π.

$$once(a, \pi) \coloneqq \neg a_\pi \,\mathcal{U}(a_\pi \wedge \bigcirc\Box\neg a_\pi)$$

We also use $once(p)$ if p is a propositionally quantified variable in HyperQPTL, i.e., when it does not refer to a specific trace. Formula $equalTrace(\pi, \pi')$ denotes that two trace variables refer to the same trace (or path).

$$equalTrace(\pi, \pi') \coloneqq \Box \bigwedge_{a \in AP} a_\pi \leftrightarrow a_{\pi'}$$

3.1.1 HyperQPTL versus FO[<, E]

As a first result, we establish the connection between FO[<, E] and HyperQPTL.

Lemma 3.1. *HyperQPTL is at least as expressive as FO[<, E].*

Proof. We give a linear translation $[\cdot]^{\mathrm{HQPTL}}$ from →FO[<, E] to →HyperQPTL such that for all trace sets T and all FO[<, E] formulas φ in prenex normal form, we obtain $T \models \varphi$

→page 29
→page 28

iff $T \models [\varphi]^{\mathrm{HQPTL}}$. We encode each first-order variable x as a combination of a trace variable π^x and a propositional variable p^x, which indicates the time component of x. We enforce p^x to hold exactly once.

$$\begin{aligned}
[P_a(x)]^{\mathrm{HQPTL}} &:= \square(p^x \rightarrow a_{\pi^x}) \\
[x < y]^{\mathrm{HQPTL}} &:= \square(p^x \rightarrow \bigcirc\Diamond p^y) \wedge \mathit{equalTrace}(\pi^x, \pi^y) \\
[x = y]^{\mathrm{HQPTL}} &:= \square(p^x \leftrightarrow p^y) \wedge \mathit{equalTrace}(\pi^x, \pi^y) \\
[E(x,y)]^{\mathrm{HQPTL}} &:= \square(p^x \leftrightarrow p^y) \\
[\neg\varphi]^{\mathrm{HQPTL}} &:= \neg[\varphi]^{\mathrm{HQPTL}} \\
[\varphi_1 \vee \varphi_2]^{\mathrm{HQPTL}} &:= [\varphi_1]^{\mathrm{HQPTL}} \vee [\varphi_2]^{\mathrm{HQPTL}} \\
[\exists x.\, \varphi]^{\mathrm{HQPTL}} &:= \exists \pi^x.\, \exists p^x.\, \mathit{once}(p^x) \wedge [\varphi]^{\mathrm{HQPTL}}
\end{aligned}$$

Since we assume φ to be in prenex normal form, $[\varphi]^{\mathrm{HQPTL}}$ can be transformed into a valid HyperQPTL formula. As a side remark, note that we could convert $[\varphi]^{\mathrm{HQPTL}}$ into a formula that only uses $\square$ and $\bigcirc$ as temporal operators. We would require that φ is in negation normal form and add the rules for $\wedge, \mathcal{W}$, and $\forall$. In the case of $x < y$, the relation between p^x and p^y can alternatively be stated as $\square(p^y \rightarrow \bigcirc\square\neg p^x)$ by exploiting the fact that p^x and p^y are required to hold exactly once. □

Theorem 3.2. *HyperQPTL is strictly more expressive than FO[$<, E$].*

Proof. With Lemma 3.1, we are left to show that there are properties that HyperQPTL can express, but FO[$<, E$] cannot. We apply a similar technique as in [195]: consider the class of single-trace trace sets. In this class, HyperQPTL is expressively equivalent to QPTL and FO[$<, E$] is expressively equivalent to FO[$<$], which is equivalent to LTL [144]. It is known, however, that QPTL is strictly more expressive than LTL since QPTL can express any ω-regular language [212], which LTL cannot [178]. □

3.1.2 S1S[E] versus HyperQPTL

We define S1S[E] and introduce HyperQPTL$^+$ as alternative definition of HyperQPTL. We subsequently show that S1S[E] and HyperQPTL$^+$ are expressively equivalent and strictly more expressive than the traditional definition of HyperQPTL.

S1S[E]. We extend →S1S with the equal-level predicate to obtain a hyperlogic. Let $V_1 = \{x_1, x_2, \ldots\}$ be a set of first-order variables and $V_2 = \{X_1, X_2, \ldots\}$ be a set of second-order variables. The syntax of S1S[E] formulas φ is defined as follows: →page 24

$$\begin{aligned}
\tau &::= x \mid \mathit{min}(x) \mid S(\tau) \\
\varphi &::= \tau \in X \mid \tau = \tau \mid E(\tau,\tau) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\, \varphi \mid \exists X.\, \varphi
\end{aligned}$$

Here, $x \in V_1$ is a first-order variable, S denotes the successor relation, and $min(x)$ indicates the minimal element of the trace addressed by x. Furthermore, $E(\tau, \tau)$ is the equal-level predicate and $X \in V_2 \cup \{X_a \mid a \in AP\}$. We interpret S1S[$E$] formulas over a set of traces T. As in the case of FO[$<, E$], the domain of the first-order variables is $T \times \mathbb{N}$. Let $\mathcal{V}_1 : V_1 \to T \times \mathbb{N}$ and $\mathcal{V}_2 : V_2 \to 2^{T \times \mathbb{N}}$ be the first-order and second-order valuation, respectively. The value of a term is defined as follows.

$$\begin{aligned}
[x]_{\mathcal{V}_1} &:= \mathcal{V}_1(x) \\
[min(x)]_{\mathcal{V}_1} &:= (\#_1(\mathcal{V}_1(x)), 0) \\
[S(\tau)]_{\mathcal{V}_1} &:= (\#_1([\tau]_{\mathcal{V}_1}), \#_2([\tau]_{\mathcal{V}_1}) + 1)
\end{aligned}$$

Let φ be an S1S[E] formula with free first-order variables $V_1' \subseteq V_1$ and free second-order variables $V_2' \subseteq V_2 \cup \{X_a \mid a \in AP\}$. We define the satisfaction relation $\mathcal{V}_1, \mathcal{V}_2 \models_T \varphi$ with respect to two valuations $\mathcal{V}_1, \mathcal{V}_2$ assigning all free variables in V_1' and V_2' as follows.

$$\begin{array}{lll}
\mathcal{V}_1, \mathcal{V}_2 \models_T \tau \in X & \text{iff} & [\tau]_{\mathcal{V}_1} \in \mathcal{V}_2(X) \\
\mathcal{V}_1, \mathcal{V}_2 \models_T \tau_1 = \tau_2 & \text{iff} & [\tau_1]_{\mathcal{V}_1} = [\tau_2]_{\mathcal{V}_1} \\
\mathcal{V}_1, \mathcal{V}_2 \models_T E(\tau_1, \tau_2) & \text{iff} & \#_2([\tau_1]_{\mathcal{V}_1}) = \#_2([\tau_2]_{\mathcal{V}_1}) \\
\mathcal{V}_1, \mathcal{V}_2 \models_T \neg\varphi & \text{iff} & \mathcal{V}_1, \mathcal{V}_2 \not\models_T \varphi \\
\mathcal{V}_1, \mathcal{V}_2 \models_T \varphi_1 \vee \varphi_2 & \text{iff} & \mathcal{V}_1, \mathcal{V}_2 \models_T \varphi_1 \text{ or } \mathcal{V}_1, \mathcal{V}_2 \models_T \varphi_2 \\
\mathcal{V}_1, \mathcal{V}_2 \models_T \exists x.\, \varphi & \text{iff} & \exists (t, n) \in T \times \mathbb{N}.\, \mathcal{V}_1[x \mapsto (t, n)], \mathcal{V}_2 \models_T \varphi \\
\mathcal{V}_1, \mathcal{V}_2 \models_T \exists X.\, \varphi & \text{iff} & \exists A \subseteq T \times \mathbb{N}.\, \mathcal{V}_1, \mathcal{V}_2[X \mapsto A] \models_T \varphi
\end{array}$$

We call an S1S[E] formula φ closed if every free variable is a second-order variable of the form X_a with $a \in AP$. We say that a trace set T over AP satisfies a closed S1S[E] formula φ, written $T \models \varphi$, if $\emptyset, \mathcal{V}_2 \models_T \varphi$, where $\emptyset$ denotes the empty first-order valuation and $\mathcal{V}_2$ assigns each free X_a in φ to the set $\{(t, i) \in T \times \mathbb{N} \mid a \in t[i]\}$.

HyperQPTL$^+$. The definition of →HyperQPTL based on QPTL is straightforward. →page 28 However, one could argue that it is not the only way to extend QPTL to a hyperlogic. The original idea of QPTL is to "color" the trace by introducing additional atomic propositions. The way HyperQPTL is defined, that idea is translated to sets of traces by coloring the traces uniformly. An alternative approach could be to color every trace individually by quantifying a standard atomic proposition for every propositional quantification. We introduce this logic as HyperQPTL$^+$ and examine is expressiveness as opposed to S1S[E] and HyperQPTL.

Similar to HyperQPTL, HyperQPTL$^+$ extends the syntax of HyperLTL with quantification over propositions. In contrast, HyperQPTL$^+$'s propositional quantification does not add a single trace over $2^{\{p\}}$ but (re)assigns an atomic proposition $a \in AP$. To distinguish between propositional quantification of HyperQPTL and HyperQPTL$^+$, we use $a, b, \ldots$ for quantified propositional variables in HyperQPTL$^+$ while we use p, q, r for uniform quantification. The syntax of the logic is given as follows. In comparison to HyperQPTL, all propositional variables are annotated with a trace variable.

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

Propositional quantification is evaluated as follows.

$$\begin{aligned}\Pi, i \models_T \exists a.\, \varphi \quad &\text{iff} \quad \exists T' \subseteq (2^{AP})^\omega, \Pi' \colon V_\pi \to T'.\, T =_{AP \setminus \{a\}} T' \text{ and } \Pi =_{AP \setminus \{a\}} \Pi' \\ &\qquad \text{and } \Pi', i \models_{T'} \varphi \\ \Pi, i \models_T \forall a.\, \varphi \quad &\text{iff} \quad \forall T' \subseteq (2^{AP})^\omega, \Pi' \colon V_\pi \to T'.\, \text{if } T =_{AP \setminus \{a\}} T' \text{ and } \Pi =_{AP \setminus \{a\}} \Pi' \\ &\qquad \text{then } \Pi', i \models_{T'} \varphi\end{aligned}$$

Note that the propositional quantification of HyperQPTL$^+$ subsumes HyperQPTL's propositional quantification, as we can express with two universal trace quantifiers that p is uniform across all traces. We show that S1S[E] and HyperQPTL$^+$ are equally expressive.

Lemma 3.3. *S1S[E] is at least as expressive as HyperQPTL$^+$.*

Proof. We describe a linear translation from HyperQPTL$^+$ to S1S[E], which employs ideas based on the translation from HyperLTL to FO[$<, E$] described in [95]. Compared with FO[$<, E$], S1S[E] does not have a $<$ operator. We thus simulate the operator with second-order quantification. Given two terms τ and τ', the following formula expresses that τ refers to an earlier point than τ'.

$$\begin{aligned}\tau < \tau' \;:=\; &\exists X.\, \tau \notin X \wedge \tau' \in X \\ &\wedge \forall x.\, (x \in X \to S(x) \in X) \wedge \forall y.\, E(x, y) \to (x \in X \leftrightarrow y \in X)\end{aligned}$$

The formula quantifies over a set of points on traces which is closed under successor. Furthermore, the smallest i for which there is a $(t, i) \in X$ is the same for all traces t. Therefore, if τ' is in the set and τ is not, then τ' must refer to a later point in time. Compared with FO[$<, E$], our definition does not enforce that τ and τ' reside on the same trace. Using S1S[E]'s native negation and equality between terms, this definition also gives us $\leq$.

We use two designated sets of first-order variables $V_1^t, V_1^i \subseteq V_1$, where we use $V_1^t = \{x_\pi, x_{\pi'}, \ldots\}$ to refer to traces and $V_1^i = \{i_1, i_2, \ldots\}$ to indicate time. Given a HyperQPTL$^+$ formula φ and a term τ that indicates the current point in time, we compositionally construct the S1S[E] formula with free second-order variables $\{X_a \mid a \in$

$AP\}$ as follows.

$$
\begin{aligned}
[a_\pi]_\tau^{\mathrm{S1S}[E]} &:= \exists x.\, min(x) = x_\pi \wedge E(x,\tau) \wedge x \in X_a\\
[\neg\varphi]_\tau^{\mathrm{S1S}[E]} &:= \neg[\varphi]_\tau^{\mathrm{S1S}[E]}\\
[\varphi_1 \vee \varphi_2]_\tau^{\mathrm{S1S}[E]} &:= [\varphi_1]_\tau^{\mathrm{S1S}[E]} \vee [\varphi_2]_\tau^{\mathrm{S1S}[E]}\\
[\bigcirc\varphi]_\tau^{\mathrm{S1S}[E]} &:= [\varphi]_{(S\ \tau)}^{\mathrm{S1S}[E]}\\
[\varphi_1 \,\mathcal{U}\, \varphi_2]_\tau^{\mathrm{S1S}[E]} &:= \exists x.\, \tau \le x \wedge [\varphi_2]_x^{\mathrm{S1S}[E]} \wedge \forall y.\, \tau \le y < x \rightarrow [\varphi_1]_y^{\mathrm{S1S}[E]}\\
[\exists\pi.\, \varphi]_\tau^{\mathrm{S1S}[E]} &:= \exists x_\pi.\, min(x_\pi) = x_\pi \wedge [\varphi]_\tau^{\mathrm{S1S}[E]}\\
[\exists a.\, \varphi]_\tau^{\mathrm{S1S}[E]} &:= \exists X_a.\, [\varphi]_\tau^{\mathrm{S1S}[E]}
\end{aligned}
$$

In the above translation, we keep the invariant that all first-order variables used to indicate traces quantify the minimum element of the trace. For a HyperQPTL$^+$ formula φ, we define

$$[\varphi]^{\mathrm{S1S}[E]} := \exists x.\, [\varphi]_{min(x)}^{\mathrm{S1S}[E]}$$

With a simple inductive argument it follows that for each HyperQPTL$^+$ formula φ and trace set T, $T \models \varphi$ iff $T \models [\varphi]^{\mathrm{S1S}[E]}$. □

Lemma 3.4. *HyperQPTL$^+$ is at least as expressive as S1S[E].*

Proof. We give a translation from an S1S[E] formula in prenex form to HyperQPTL$^+$.

$$
\begin{aligned}
[\tau \in X]^{\mathrm{HQPTL}^+} &:= isMember(\tau, a^X)\\
[\tau_1 = \tau_2]^{\mathrm{HQPTL}^+} &:= isEqual(\tau_1, \tau_2)\\
[E(\tau_1, \tau_2)]^{\mathrm{HQPTL}^+} &:= isLevel(\tau_1, \tau_2)\\
[\neg\varphi]^{\mathrm{HQPTL}^+} &:= \neg[\varphi]^{\mathrm{HQPTL}^+}\\
[\varphi_1 \vee \varphi_2]^{\mathrm{HQPTL}^+} &:= [\varphi_1]^{\mathrm{HQPTL}^+} \vee [\varphi_2]^{\mathrm{HQPTL}^+}\\
[\exists x.\, \varphi]^{\mathrm{HQPTL}^+} &:= \exists \pi^x.\, \exists i^x.\, once(i^x, \pi^x) \wedge [\varphi]^{\mathrm{HQPTL}^+}\\
[\exists X.\, \varphi]^{\mathrm{HQPTL}^+} &:= \exists a^X.\, [\varphi]^{\mathrm{HQPTL}^+}
\end{aligned}
$$

We use HyperQPTL$^+$'s second-order quantification to denote a point on a trace. While the quantification of i^x re-assigns the proposition on all traces, we only refer to it on trace π^x and therefore only require that it holds exactly once on π^x. To translate S1S[E]'s terms, we use predicates *isMember*, *isEqual*, and *isLevel*, which straightforwardly unpack the terms as follows. The predicate $isMember(\tau, a^X)$ checks if proposition a^X holds on the trace and point in time denoted by τ.

$$
\begin{aligned}
isMember(S^n(min(x)), a^X) &:= \bigcirc^n a^X_{\pi^x}\\
isMember(S^n(x), a^X) &:= \square(i^x_{\pi^x} \rightarrow \bigcirc^n a^X_{\pi^x})
\end{aligned}
$$

The predicate $isLevel(\tau, \tau')$ checks if two terms refer to the same point in time.

$$\begin{array}{lll}
isLevel(S^n(min(x)), S^n(min(y))) & := & true \\
isLevel(S^m(min(x)), S^n(y)) & := & \bigcirc^{m-n}\, i^y_{\pi^y} \qquad \text{if } n \leq m \\
isLevel(S^m(x), S^n(min(y))) & := & \bigcirc^{n-m}\, i^x_{\pi^x} \qquad \text{if } m \leq n \\
isLevel(S^m(x), S^n(y)) & := & \begin{cases} \square(i^x_{\pi^x} \rightarrow \bigcirc^{m-n}\, i^y_{\pi^y}) & \text{if } n \leq m \\ \square(i^y_{\pi^y} \rightarrow \bigcirc^{n-m}\, i^x_{\pi^x}) & \text{if } m \leq n \end{cases} \\
isLevel(\tau_1, \tau_2) & := & false \qquad \text{otherwise}
\end{array}$$

Finally, the predicate $isEqual(\tau, \tau')$ checks if two terms refer to the same trace and point in time.

$$isEqual(\tau_1, \tau_2) \;:=\; equalTrace(\pi^x, \pi^y) \wedge isLevel(\tau_1, \tau_2)$$

where the base variable of τ_1 is x, and the one of τ_2 is y. Now, let φ be a closed S1S[E] formula. The above translation shows that for any set of traces $T \subseteq (2^{AP})^\omega$, $T \models \varphi$ iff $T \models [\varphi]^{\text{HQPTL}^+}$. □

From the above two lemmas we obtain the following theorem.

Theorem 3.5. *HyperQPTL⁺ and S1S[E] are equally expressive.*

We establish that in contrast to HyperQPTL [195], the model checking problem of HyperQPTL$^+$ and S1S[E] is undecidable.

Theorem 3.6. *The S1S[E] model checking problem is undecidable.*

Proof. The S1S[E] model checking problem is to decide for a formula φ and transition system $\mathcal{S}$, if $traces(\mathcal{S}) \models \varphi$. We prove this lemma by a reduction from →deterministic 2-counter machines. We describe a simple trace set T such that given a 2CM $\mathcal{M}$ with an initial configuration s_0, we can construct an S1S[E] formula $\varphi_{\mathcal{M},s_0}$ such that $\mathcal{M}$ halts on s_0 iff $T \models \varphi_{\mathcal{M},s_0}$. We spell out the main ideas of the reduction. First, we note that we can express in S1S[E] that a set X_s contains exactly the nodes of a single infinite trace. →page 21

$$trace(X_s) := \exists x.\, x \in X_s \wedge \forall y.\, y \in X_s \leftrightarrow min(y) = min(x)$$

Each 2CM configuration s is encoded as a trace over atomic propositions c_1, c_2, and l which are all true exactly once on the trace. A state where the first counter has value v is encoded as a trace t with $c_1 \in t[v]$. If l is true at position i, the next instruction to be executed is instruction l_i. If the 2CM has m instructions, l cannot appear later than at position $m-1$. We call the S1S[E] formula encoding that a set X_s encodes a valid configuration $config(X_s)$. We choose T to be the infinite set of traces containing the trace encoding for all configurations $s \in \{(i, v_1, v_2) \in \mathbb{N}^3 \mid 0 \leq i \leq m\}$, where m is the number of instructions of the 2CM. Note that T is clearly producible by a finite transition system. We can give an S1S[E] formula $succ_i(X_s, X_{s'})$, which is true iff the next instruction to be executed in s is instruction l_i and configuration s' is the single next configuration. We give an exemplary formula for the case that l_i is of the second instruction type and that the first counter is tested for 0.

$$\begin{aligned} succ_i(X_s, X_{s'}) \coloneqq \forall x \in X_s, y \in X_{s'}.\, & S^i(min(x)) \in X_l \\ & \wedge (min(x) \in X_{c_1} \rightarrow S^j(min(y)) \in X_l \\ & \quad \wedge (E(x,y) \rightarrow (x \in X_{c_1} \leftrightarrow y \in X_{c_1}) \wedge (x \in X_{c_2} \leftrightarrow y \in X_{c_2}))) \\ & \wedge (min(x) \notin X_{c_1} \rightarrow S^k(min(y)) \in X_l \\ & \quad \wedge (E(x,y) \rightarrow (S(x) \in X_{c_1} \leftrightarrow y \in X_{c_1}) \wedge (x \in X_{c_2} \leftrightarrow y \in X_{c_2}))) \end{aligned}$$

Given a machine $\mathcal{M}$ with initial configuration s_0, we give an S1S[E] formula $halting(X)$ which is true iff the set of traces X encodes a sequence of configurations that form a halting computation in $\mathcal{M}$. The formula $halting(X)$ is a conjunct of the following requirements:

- X is a union of finitely many encoded configurations X_s. We express this in S1S[E] by stating that X contains only full traces that encode valid configurations and that there is an upper bound on the points in X where c_1 and c_2 occur.

$$\begin{aligned} & \forall x \in X.\, min(x) \in X \wedge S(x) \in X \wedge (\forall X_s \subseteq X.\, trace(X_s) \rightarrow config(X_s)) \\ & \wedge \exists x \in X. \forall y \in X. x < y \rightarrow y \notin X_{c_1} \wedge y \notin X_{c_2} \end{aligned}$$

- X is predecessor closed with respect to the instructions of the machine, i.e., if X_s is a configuration in X, then either X_s encodes the initial configuration s_0, or there is a $X_{s'} \subseteq X$ such that $succ_i(X_{s'}, X_s)$ for some i.

$$\begin{aligned} & \forall X_s \subseteq X.\, trace(X_s) \wedge config(X_s) \rightarrow \\ & \quad \exists X_{s'} \subseteq X.\, trace(X_{s'}) \wedge config(X_{s'}) \wedge \bigvee_{1 \leq i \leq m} succ_i(X_s, X_{s'}) \end{aligned}$$

- There is the encoding of a halting configuration X_{halt} in X.

Using the ideas presented above, we can define $\varphi_{\mathcal{M},s_0}$ as a formula that checks whether there is a subset X of T which encodes a halting computation of $\mathcal{M}$ starting in s_0, i.e., $\varphi_{\mathcal{M},s_0} \coloneqq \exists X.\, halting(X)$. □

Finally, we can state the last theorem that completes the linear-time hierarchy.

Theorem 3.7. *S1S[E] and HyperQPTL$^+$ are strictly more expressive than HyperQPTL.*

Proof. As HyperQPTL$^+$ trivially subsumes HyperQPTL, this follows from Theorem 3.5 and Theorem 3.6, since the HyperQPTL model checking problem is decidable [195]. □

3.1.3 A Case for HyperQPTL

In many aspects, HyperQPTL can be seen as the sweet spot in the linear-time hierarchy of hyperlogics. As opposed to HyperQPTL$^+$ and S1S[E], the model checking problem of HyperQPTL is still decidable [195] (which is a strong argument in the context of formal methods). Besides the fact that HyperQPTL is strictly more expressive than FO[$<, E$], HyperQPTL formulas also need fewer quantifiers to express the same statement. Temporal operators are algorithmically easier to handle than first-order quantifiers, a reason the success of LTL over first-order logic. This is also makes HyperQPTL formulas oftentimes easier to read than their FO[$<, E$] equivalent.

As an illustration, consider the class of promptness properties, e.g., *bounded waiting for a grant.* The property states that if some agent requests access to a shared resource at point in time i, then it will be granted access within a bounded amount of time. The bound may depend on the point in time i where access to the resource was requested. However, it may not depend on the current trace. We express this property in HyperQPTL as follows.

$$\forall i.\, \exists b.\, \forall \pi.\, \square(i \wedge req_\pi \rightarrow \diamondsuit b \wedge (\neg b \,\mathcal{U}\, grant_\pi))$$

The formula states that for every point in time i, there exists a bound b such that if a trace requires access at point in time i, then it will not have to wait longer than until the next occurrence of b. Note that this property differs from saying "all traces are eventually granted access", where the bound may also depend on the trace under consideration. In that scenario, each of the infinitely many traces could wait arbitrarily long for the grant. In particular, it could happen that with each trace, the waiting time is longer than before. While this formula could be translated to FO[$<, E$] (t and b are not ω-regular, they just quantify points in time), the formulation in HyperQPTL focuses on the crucial alternation of the type of quantifier: first quantify the points in time, then quantify the trace. In HyperQPTL, we do not need the quantifiers that FO[$<, E$] would need for the temporal operators.

3.2 TeamLTL in the Linear-time Hierarchy

Temporal team logics constitute a new, fundamentally different approach to specifying hyperproperties. Team logics argue over teams; in the case of TeamLTL, the team is a set of traces. The syntax of LTL with team semantics is that of LTL but the semantics differ. In LTL with team semantics, $\vee$ serves as a split operator, which splits the team (i.e., the trace set) during the evaluation of a formula. This enables us to express properties of subsets of traces. As an example, consider the case that an unknown input determines the behavior of the system. Depending on the input, its execution traces either agree on a or on b. We can express the property in HyperLTL with three trace quantifiers:

$$\exists \pi_1, \pi_2. \forall \pi. \square(a_{\pi_1} \leftrightarrow a_\pi) \vee \square(b_{\pi_2} \leftrightarrow b_\pi)$$

In TeamLTL, the same property can be simply expressed as

$$\square(a \varovee \neg a) \vee \square(b \varovee \neg b)$$

The Boolean or operator $\varovee$ expresses that in the current team, either the left side holds on all traces or the right side does. The use of the $\varovee$ operator reveals another strength of TeamLTL: its modularity. The research on team semantics (see →Chapter 5 for related work) has a rich tradition of studying extensions of team logics with new atomic statements and operators. They constitute a well-defined way to increase a logic's expressiveness in a step-by-step manner. Besides $\varovee$, examples are Boolean negation $\sim$, the inclusion atom $\subseteq$, and universal subteam quantifiers A and A^1. Inclusion atoms have been found to be fascinating for their ability to express recursion in the first-order setting [110]. The operator A^1 can express all LTL-definable properties. With the introduction of generalized atoms, TeamLTL even permits custom extensions. →page 71

Possibly most interesting in the context of hyperproperties are dependence atoms. A dependence atom $\mathrm{dep}(x_1, \ldots, x_n)$ is satisfied by a team X if any two assignments assigning the same values to the variables $x_1, \ldots, x_{n-1}$ also assign the same value to x_n. For example, the TeamLTL formula

$$(\square\, \mathrm{dep}(i_1, i_2, o)) \vee (\square\, \mathrm{dep}(i_2, i_3, o))$$

states that the set of traces can be decomposed into two parts; in the first part, the output o is determined by the inputs i_1 and i_2, and in the second part, o is determined by the inputs i_2 and i_3.

Only very few results are known about the expressive power of temporal logics with team semantics. This applies, in particular, to how TeamLTL compares to quantifier-based temporal hyperlogics. What is known is that HyperLTL and TeamLTL are incomparable in expressiveness [153]. In this section, we identify two extensions of TeamLTL that can express all (and all downward closed, respectively) Boolean relations on LTL properties of teams, and present translations from these logics to HyperQPTL and HyperQPTL$^+$.

3.2.1 Expressive Extensions of TeamLTL

First, we recap some typical properties associated with logics with team semantics.

Downward closure property. If $T, i \models \varphi$ and $T' \subseteq T$, then $T', i \models \varphi$.

Empty team property. $\emptyset, i \models \varphi$

Flatness property. $T, i \models \varphi$ iff $\forall t \in T.\, \{t\}, i \models \varphi$

Singleton equivalence property. $\{t\}, i \models \varphi$ iff $t, i \models \varphi$

In the singleton equivalence property, the rightmost satisfaction relation $\models$ is that of standard LTL. A team logic has one of the above properties if every formula of the logic has the property. TeamLTL satisfies downward closure, singleton equivalence, and the empty team property [153]. TeamLTL does not satisfy flatness; for instance, the formula $\Diamond p$ is not flat [153].

We define extensions of TeamLTL that are typical for logics with team semantics, i.e., dependence atoms, inclusion atoms, subteam quantifiers, and Boolean negation and disjunction. We then introduce LTL-definable generalized atoms for TeamLTL and present the two extensions of TeamLTL we use in the rest of the section.

Dependence atoms are of the form $\mathrm{dep}(\psi_1, \dots, \psi_n, \psi)$ and inclusion atoms are given as $\psi_1, \dots, \psi_n \subseteq \psi'_1, \dots, \psi'_n$, where all ψ, ψ_i, and ψ'_i are LTL formulas interpreted with the standard semantics of LTL. Dependence atoms state that the value of ψ for team (T, i) is functionally determined by that of $\psi_1, \dots, \psi_n$. For TeamLTL, they were first defined in [153]. Inclusion atoms state that each value combination of $\psi_1, \dots, \psi_n$ must also occur as a value combination for $\psi'_1, \dots, \psi'_n$. Their formal semantics is defined as:

$$\begin{array}{lll} T, i \models \mathrm{dep}(\psi_1, \dots, \psi_n, \psi) & \text{iff} & \forall t, t' \in T.\ \text{if} \bigwedge\limits_{1 \le j \le n} (t, i \models \psi_j \text{ iff } t', i \models \psi_j) \\ & & \qquad\qquad\text{then } (t, i \models \psi \text{ iff } t', i \models \psi) \\ T, i \models \psi_1, \dots, \psi_n \subseteq \psi'_1, \dots, \psi'_n & \text{iff} & \forall t \in T.\, \exists t' \in T.\ \bigwedge\limits_{1 \le j \le n} t, i \models \psi_j \text{ iff } t', i \models \psi'_j \end{array}$$

As an example, let $o_1, \dots, o_n$ be some observable outputs and s be a secret. The atom $(o_1, \dots, o_n, s) \subseteq (o_1, \dots, o_n, \neg s)$ expresses a form of *noninference* by stating that an observer cannot infer the current value of the secret from the (observable) outputs. We also consider other connectives commonly used for logics with team semantics: *Boolean disjunction* $\varovee$, *Boolean negation* $\sim$, and *universal subteam quantifiers* A and A^1, with their semantics defined as:

$$\begin{array}{lll} T, i \models \psi_1 \varovee \psi_2 & \text{iff} & T, i \models \psi_1 \text{ or } T, i \models \psi_2 \\ T, i \models {\sim}\psi & \text{iff} & T, i \not\models \psi \end{array}$$

$$\begin{aligned} T, i &\models \mathsf{A}\psi && \text{iff} \quad \forall T' \subseteq T.\, T', i \models \psi \\ T, i &\models \mathsf{A}^1\psi && \text{iff} \quad \forall t \in T.\, \{t\}, i \models \psi \end{aligned}$$

While $\vee$ under team semantics expresses that the trace set can be split into two sets, $\varovee$ states that either the whole set satisfies ψ_1 or the whole set satisfies ψ_2. The operator A enforces that formula ψ is downward closed as in the downward closedness property, and A^1 enforces flatness. If $\mathcal{A}$ is a set of non-standard operators, we denote by TeamLTL($\mathcal{A}$) the extension of TeamLTL with the operators in $\mathcal{A}$. For any operator $\circ$, we simply write TeamLTL($\mathcal{A}, \circ$) instead of TeamLTL($\mathcal{A} \cup \{\circ\}$).

TeamLTL($\sim$) is a very expressive logic; all of the above operators have been shown to be expressible in TeamLTL($\sim$) [130, 168]. To systematically explore variants of TeamLTL that fall between plain TeamLTL and TeamLTL($\sim$), we employ two representative logics, namely TeamLTL($\varovee, \mathsf{A}^1$) and TeamLTL($\varovee, \mathsf{A}^1, \sim\bot$). Here, $\sim\bot$ is syntactic sugar for $\sim$*false*, which enforces that the trace set T contains at least one trace. We therefore also call it the non-empty-team operator. These logics form an interesting extension of TeamLTL: they are more expressive than plain TeamLTL and less expressive than TeamLTL($\sim$). But most importantly, we show that they can express a general class of Boolean relations over teams (T, i). To do so, we describe the notion of LTL-definable *generalized atoms* for TeamLTL.

These atoms were first introduced, in the first-order team semantics setting, by Kuusisto [156] using generalized quantifiers. First-order-definable generalized atoms for TeamLTL were also briefly mentioned in [153].

Definition 3.1 (Generalized atoms for LTL)**.** An n-ary generalized atom $\#_G(\psi_1, \ldots, \psi_n)$ is a tuple of n LTL formulas $\psi_1, \ldots, \psi_n$ together with a set G of n-ary relations over the Boolean domain. Its team semantics is defined as follows.

$$T, i \models \#_G(\psi_1, \ldots, \psi_n) \quad \text{iff} \quad \{((t, i \models \psi_1), \ldots, (t, i \models \psi_n)) \mid t \in T\} \in G$$

Generalized atoms describe properties of teams (T, i) (think of a team as a "trace set slice") by reducing each trace to its valuation on some LTL properties and then stating which valuations are allowed to occur together in a slice. One of the simplest generalized atoms is the unary atom $\#_G(a)$ with $G = \{\{\mathit{true}\}, \{\mathit{false}\}, \emptyset\}$, which is true for trace set T at position i if all traces in T agree on the value of a at position i. The empty set is necessary in G for the case that T is empty. Similarly, $\mathsf{A}^1\psi$ can be interpreted as a unary generalized atom with $G = \{\{\mathit{true}\}, \emptyset\}$. For each arity n, we can also define the n-ary dependence atom as generalized atom. The binary dependence atom $\mathrm{dep}(\psi, \psi')$ can be expressed as generalized atom $\#_G(\psi, \psi')$ with $G = \{A \subseteq \{0, 1\}^2 \mid$ if $(a, b_1) \in A$ and $(a, b_2) \in A$ then $b_1 = b_2\}$.

A generalized atom $\#_G(\psi_1, \ldots, \psi_n)$ is downward closed if G is downward closed, i.e., if $R \in G$ and $R' \subseteq R$, then $R' \in G$. We denote by $\mathcal{A}_{\mathrm{all}}$ and $\mathcal{A}_{\mathrm{dc}}$ the set of all and all generalized atoms and the set of all downward-closed generalized atoms. Using a straightforward induction we can prove that for any set $\mathcal{A}$ of downward closed atoms and operators (an operator is downward closed if it preserves downward closedness), the logic TeamLTL($\mathcal{A}$) is downward closed as well. For instance, TeamLTL(dep, $\varovee$, A^1) is downward closed.

The next proposition establishes that TeamLTL($\varovee$, A^1) can express all downward-closed generalized atoms. Additionally, adding $\sim\perp$ to the logic is sufficient to express all generalized atoms. The translation given in the proposition is inspired by an analogous one given in [231] for propositional team logics.

Proposition 3.8. *TeamLTL($\varovee$, A^1) is expressively equivalent to TeamLTL($\mathcal{A}_{\mathrm{dc}}$, $\varovee$), and TeamLTL($\varovee$, A^1, $\sim\perp$) is expressively equivalent toTeamLTL($\mathcal{A}_{\mathrm{all}}$, $\varovee$).*

Proof. First, note that the size of G and the contained relations are by construction finite. For any $R \in G$ and $(b_1, \ldots, b_n) \in R$, let $b := (b_1, \ldots, b_n)$ and $\vec{\psi}^b := \psi_1^{b_1} \wedge \ldots \wedge \psi_n^{b_n}$, where we define $\psi_i^{true} := \psi_i$ and $\psi_i^{false} := \neg\psi_i$ in negation normal form. We prove that for any n-ary generalized atom $\#_G(\psi_1, \ldots, \psi_n)$, we have that

$$\#_G(\psi_1, \ldots, \psi_n) \equiv \bigovee_{R\in G} \bigvee_{b\in R} \mathsf{A}^1 \left(\vec{\psi}^b \wedge \sim\perp\right)$$

If $\#_G$ is downward closed, we show that the above translation can be simplified, removing the need for the non-empty-team operator.

$$\#_G(\psi_1, \ldots, \psi_n) \ \equiv \ \bigovee_{R\in G} \bigvee_{b\in R} \mathsf{A}^1 \vec{\psi}^b$$

Let $[\![\vec{\psi}\,]\!]_{(t,i)} := ((t, i \models \psi_1), \ldots, (t, i \models \psi_n))$ for any trace t and $i \in \mathbb{N}$. First, we observe that for every trace set T and $i \in \mathbb{N}$

$$\begin{aligned} T, i \models \mathsf{A}^1 \vec{\psi}^b \ &\Leftrightarrow \ T = \emptyset \text{ or } \forall t \in T.\, [\![\vec{\psi}]\!]_{(t,i)} = b \\ &\Leftrightarrow \ \{[\![\vec{\psi}]\!]_{(t,i)} \mid t \in T\} \subseteq \{b\} \end{aligned}$$

Thus, due to the team semantics of the $\vee$ operator,

$$T, i \models \bigvee_{b\in R} \mathsf{A}^1 \vec{\psi}^b \ \Leftrightarrow \ \{[\![\vec{\psi}]\!]_{(t,i)} \mid t \in T\} \subseteq R$$

If we add the requirement that every team generated by every disjunct above is non-empty, we get

$$T, i \models \bigvee_{b\in R} \left(\mathsf{A}^1 \vec{\psi}^b \wedge \sim\perp\right) \ \Leftrightarrow \ \{[\![\vec{\psi}]\!]_{(t,i)} \mid t \in T\} = R.$$

Therefore, if $\#_G(\psi_1, \dots, \psi_n)$ is downward closed, then

$$\begin{aligned}
&T, i \models \bigovee_{R \in G} \bigvee_{b \in R} \mathsf{A}^1 \vec{\psi}^b \\
\Leftrightarrow\ & \{[\![\vec{\psi}\,]\!]_{(t,i)} \mid t \in T\} \subseteq R && \text{for some } R \in G \text{, by def. of } \ovee \text{ and reasoning above} \\
\Leftrightarrow\ & \{[\![\vec{\psi}\,]\!]_{(t,i)} \mid t \in T\} = R' && \text{for some } R' \subseteq R \\
\Leftrightarrow\ & T, i \models \#_G(\psi_1, \dots, \psi_n) && \text{as } G \text{ is downward closed, so } R' \in G
\end{aligned}$$

If $\#_G(\psi_1, \dots, \psi_n)$ is not downward closed, then we get

$$\begin{aligned}
&T, i \models \bigovee_{R \in G} \bigvee_{b \in R} (\mathsf{A}^1 \vec{\psi}^b \wedge {\sim}\bot) \\
\Leftrightarrow\ & \{[\![\vec{\psi}\,]\!]_{(t,i)} \mid t \in T\} = R && \text{for some } R \in G \text{ by reasoning above} \\
\Leftrightarrow\ & T, i \models \#_G(\psi_1, \dots, \psi_n) && \text{by Definition 3.1 since } R \in G
\end{aligned}$$

This concludes the proof. □

3.2.2 TeamLTL versus HyperQPTL$^+$

We relate the expressiveness of TeamLTL($\ovee$, A^1) and TeamLTL($\ovee$, A^1, ${\sim}\bot$) to fragments of HyperQPTL$^+$. As discussed in Section 3.1.2, HyperQPTL$^+$ can simulate uniform propositional quantification. We describe fragments of HyperQPTL$^+$ by restricting the quantifier prefixes of formulas. We use $\exists_\pi$ / $\forall_\pi$ to denote trace quantification, $\exists_p$ / $\forall_p$ for uniform propositional quantification, and $\exists_+$ / $\forall_+$ for non-uniform propositional quantification. We use $\exists$ ($\forall$, respectively) if we do not need to distinguish between the different types of existential (universal, respectively) quantifiers. As before, we also write Q to refer to both $\exists$ and $\forall$. As an example, $\forall^*\exists^*$HyperQPTL$^+$ refers to HyperQPTL$^+$ formulas with quantifier prefix $\{\forall_p, \forall_+, \forall_\pi\}^*\{\exists_p, \exists_+, \exists_\pi\}^*$.

We show that TeamLTL($\ovee$, A^1) and TeamLTL($\ovee$, A^1, ${\sim}\bot$) can be translated to the prefix fragments $\exists_+ Q_p^* \exists_\pi^* \forall_\pi$ and $\exists_+ Q_p^* \forall_\pi$ of HyperQPTL$^+$. The translations provide insight into the limits of the expressiveness of different extensions of TeamLTL. In particular, they show that one existential second-order quantifier $\exists_+$ is sufficient to simulate the generation of subteams with the $\vee$ operator. The difference between downward-closed team properties and general team properties manifests itself by a different need for trace quantifiers: for downward-closed properties, a single $\forall_\pi$ quantifier is enough, whereas in the general case, we need a $\exists_\pi^* \forall_\pi$ quantifier alternation.

As a prerequisite for the translation, we formulate two lemmas. The first lemma states that for every TeamLTL($\subseteq$, $\ovee$, A^1, ${\sim}\bot$) formula ψ, the formula $\mathsf{A}^1\psi$ can be treated as an LTL formula. This result is not surprising, as the A^1 operator evaluates ψ individually for every trace in the team. Proving the result requires a careful construction, however. As an intermediate step, we prove that we can remove any occurrence of $\subseteq$, $\otimes$, A^1, and $\sim\bot$ from ψ such that it remains equivalent under LTL semantics.

Lemma 3.9. *Given a TeamLTL($\subseteq, \otimes, \mathsf{A}^1, \sim\bot$) formula ψ, we can construct a TeamLTL formula ψ^* such that for all traces t and $i \in \mathbb{N}$, $\{t\}, i \models \psi$ iff $\{t\}, i \models \psi^*$.*

Proof. To construct ψ^*, we describe a function *elim*, which for TeamLTL($\subseteq, \otimes, \mathsf{A}^1, \sim\bot$) formula ψ, eliminates all occurrences of $\subseteq$, $\otimes$, A^1, and $\sim\bot$ such that for every t, we have $\{t\}, i \models \mathit{elim}(\psi)$ iff $\{t\}, i \models \psi$. Handling operators $\subseteq$ and A^1 is simple: as both sides of $\subseteq$ contain LTL formulas, we can replace $\subseteq$ with a simple equivalence; occurrences of A^1 can just be dropped. If both subformulas do not contain a $\sim\bot$ expression, then $\psi_1 \otimes \psi_2$ can just be replaced with $\psi_1 \vee \psi_2$. The main difficulty is to translate the $\vee$ operator in combination with $\sim\bot$. During the recursive evaluation of the formula, the current team might be empty because of a split, which causes $\sim\bot$ to evaluate to *false*. The idea is to "bubble up" any occurrences of $\sim\bot$ in ψ. The result of $\mathit{elim}(\psi)$ is either a formula θ or a formula $\theta \wedge \sim\bot$ such that θ is a TeamLTL formula. In the construction below, we use $\mathit{elim}(\psi) = \theta$ to state that $\mathit{elim}(\psi)$ does not contain a $\sim\bot$ expression. In an expression $\theta \wedge \sim\bot$, θ might also be empty, i.e., equal to *true*.

$$
\begin{aligned}
&\mathit{elim}(\psi_1, \dots, \psi_n \subseteq \psi'_1, \dots, \psi'_n) := \textstyle\bigwedge_{1 \le i \le n} \psi_i \leftrightarrow \psi'_i \\
&\mathit{elim}(\psi) := \psi \qquad \text{if } \psi = a, \neg a, \text{ or } \sim\bot \\
&\mathit{elim}(\mathsf{A}^1 \psi) := \mathit{elim}(\psi)
\end{aligned}
$$

$$
\mathit{elim}(\psi_1 \vee \psi_2) := \begin{cases} \sim\bot \wedge \theta_1 \wedge \theta_2 & \text{if } \mathit{elim}(\psi_1) = \sim\bot \wedge \theta_1 \text{ and } \mathit{elim}(\psi_2) = \sim\bot \wedge \theta_2 \\ \sim\bot \wedge \theta_1 & \text{if } \mathit{elim}(\psi_1) = \sim\bot \wedge \theta_1 \text{ and } \mathit{elim}(\psi_2) = \theta_2 \\ \theta_1 \vee \theta_2 & \text{else} \end{cases}
$$

$$
\mathit{elim}(\psi_1 \otimes \psi_2) := \begin{cases} \sim\bot \wedge (\theta_1 \vee \theta_2) & \text{if } \mathit{elim}(\psi_1) = \sim\bot \wedge \theta_1 \text{ and } \mathit{elim}(\psi_2) = \sim\bot \wedge \theta_2 \\ \theta_1 \vee \theta_2 & \text{else} \end{cases}
$$

$$
\mathit{elim}(\psi_1 \wedge \psi_2) := \begin{cases} \sim\bot \wedge \theta_1 \wedge \theta_2 & \text{if } \mathit{elim}(\psi_1) = \sim\bot \wedge \theta_1 \text{ or } \mathit{elim}(\psi_2) = \sim\bot \wedge \theta_2 \\ \theta_1 \wedge \theta_2 & \text{else} \end{cases}
$$

$$
\mathit{elim}(\bigcirc \psi) := \begin{cases} \sim\bot \wedge \bigcirc \theta & \text{if } \mathit{elim}(\psi) = \sim\bot \wedge \theta \\ \bigcirc \theta & \text{else} \end{cases}
$$

$$
\mathit{elim}(\psi_1 \mathcal{U} \psi_2) := \begin{cases} \sim\bot \wedge (\theta_1 \mathcal{U} \theta_2) & \text{if } \mathit{elim}(\psi_2) = \sim\bot \wedge \theta_2 \\ \theta_1 \mathcal{U} \theta_2 & \text{else} \end{cases}
$$

$$
\mathit{elim}(\psi_1 \mathcal{W} \psi_2) := \begin{cases} \sim\bot \wedge (\theta_1 \mathcal{W} \theta_2) & \text{if } \mathit{elim}(\psi_1) = \sim\bot \wedge \theta_1 \text{ and } \mathit{elim}(\psi_2) = \sim\bot \wedge \theta_2 \\ \theta_1 \mathcal{W} \theta_2 & \text{else} \end{cases}
$$

The above translation preserves the truth value of formulas over trace sets of cardinality at most 1. On singleton trace sets, θ and $\sim\!\bot \wedge \theta$ are equivalent. Thus, for both cases, if $elim(\psi) = \theta$ or if $elim(\psi) = \sim\!\bot \wedge \theta$, we set $\psi^* = \theta$ and the statement follows. □

As TeamLTL satisfies the singleton equivalence property, we obtain the following corollary from the above lemma.

Corollary 3.10. *Given a TeamLTL$(\subseteq, \otimes, \mathsf{A}^1, \sim\!\bot)$ formula ψ, we can construct an LTL formula ψ^* such that for all traces t and $i \in \mathbb{N}$, $\{t\}, i \models \psi$ iff $t, i \models \psi^*$.*

As a second prerequisite for the translation, we show that the evaluation of a TeamLTL$(\otimes, \mathsf{A}^1, \sim\!\bot)$ formula can only create countably many different teams, even if the formula is evaluated on an uncountable trace set. We show that for every set of traces T, there is a countable set $\mathcal{H} \subseteq 2^T$ such that the split operator $\vee$ only chooses trace sets from $\mathcal{H}$, independently of the formula ψ.

Lemma 3.11. *For every set T of traces over AP, there exists a countable $\mathcal{H} \subseteq 2^T$ such that for every TeamLTL$(\otimes, \mathsf{A}^1, \sim\!\bot)$ formula ψ and $i \in \mathbb{N}$,*

$$T, i \models \psi \quad \textit{iff} \quad T, i \models_{\mathcal{H}} \psi$$

where $\models_{\mathcal{H}}$ is defined such that in the evaluation of $\vee$, the subsets T_1 and T_2 are both required to be in $\mathcal{H}$.

Proof. First note that we cannot choose $\mathcal{H}$ as the set of all subsets of T, as this set might be uncountable. We therefore inductively define a nondeterministic function *traceSets*, which takes a trace set T, a natural number i, and a TeamLTL$(\otimes, \mathsf{A}^1, \sim\!\bot)$ formula ψ and returns a set of sets of traces $\mathcal{H}$ as follows.

$$\begin{aligned}
\mathit{traceSets}(T, i, \psi) &:= \{T\} \quad (\text{if } \psi = a, \neg a, \sim\!\bot, \text{ or } \mathsf{A}^1 \psi')\\
\mathit{traceSets}(T, i, \bigcirc \psi) &:= \mathit{traceSets}(T, i+1, \psi)\\
\mathit{traceSets}(T, i, \psi_1 \wedge \psi_2) &:= \mathit{traceSets}(T, i, \psi_1) \cup \mathit{traceSets}(T, i, \psi_2)\\
\mathit{traceSets}(T, i, \psi_1 \otimes \psi_2) &:= \mathit{traceSets}(T, i, \psi_1) \cup \mathit{traceSets}(T, i, \psi_2)\\
\mathit{traceSets}(T, i, \psi_1 \vee \psi_2) &:= \mathit{traceSets}(T_1, i, \psi_1) \cup \mathit{traceSets}(T_2, i, \psi_2) \cup \{T\}
\end{aligned}$$

(where T_1 and T_2 are guessed nondeterministically such that $T_1 \cup T_2 = T$)

$$\mathit{traceSets}(T, i, \psi_1 \circ \psi_2) := \bigcup_{j \geq i} \big(\mathit{traceSets}(T, j, \psi_1) \cup \mathit{traceSets}(T, j, \psi_2)\big)$$

where $\circ = \mathcal{U}$ or $\circ = \mathcal{W}$. As the countable union of countable sets is countable, any set generated by *traceSets*(T, i, ψ) is countable. Now, we choose

$$\mathcal{H} := \bigcup_{\substack{i \in \mathbb{N} \\ \psi \in \mathrm{TeamLTL}(\otimes, \mathsf{A}^1, \sim\bot)}} \mathit{traceSets}(T, i, \psi)$$

First, $T, i \models \psi$ iff $T, i \models_{\mathcal{H}} \psi$ holds for the above $\mathcal{H}$ by the definition of *traceSets*. As $\mathbb{N}$ and the formulas of TeamLTL($\otimes$, A^1, $\sim\bot$) are both countable sets, $\mathcal{H}$ is countable. The definition of *traceSets* is nondeterministic, thus $\mathcal{H}$ defines possibly uncountable many sets simultaneously. The proof is therefore not constructive, it just proves the existence of a suitable set, depending on how the trace set has to be split in case of $\vee$ to match the evaluation of $T, i \models \psi$. □

Having established the necessary lemmas, we can now prove that HyperQPTL$^+$ subsumes TeamLTL($\otimes$, A^1, $\sim\bot$).

Theorem 3.12. *For every $\psi \in$ TeamLTL($\otimes$, A^1, $\sim\bot$), there exists an equivalent formula φ in the $\exists_+ Q_p^* \exists_\pi^* \forall_\pi$ fragment of HyperQPTL$^+$. If $\psi \in$ TeamLTL($\otimes$, A^1), φ can be defined in the $\exists_+ Q_p^* \forall_\pi$ fragment.*

Proof. Let $a^{\mathcal{H}}$, p, and r be distinct propositional variables. We define a compositional translation $[\cdot]^{\text{HQPTL}^+}_{(p,r)}$ such that for every team (T, i) and TeamLTL($\otimes$, A^1, $\sim\bot$) formula ψ, we obtain

$$T, i \models \psi \quad \text{iff} \quad \emptyset, i \models_T \exists a^{\mathcal{H}}. \exists p, r. [\psi]^{\text{HQPTL}^+}_{(p,r)} \wedge \forall \pi. a^{\mathcal{H}}_\pi \wedge p_\pi \wedge r_\pi$$

Note that by our convention formulated in the definition of HyperQPTL$^+$ (cf. Section 3.1.2), $\exists a^{\mathcal{H}}$ is the normal quantification of HyperQPTL$^+$, while all other propositional variables are uniformly quantified. We require that the uniformly quantified propositional variables p and r satisfy the formula →*once* for trace π. This can be easily added to the formula, so we don't make it explicit for readability. The idea behind the translation is the following. Let T' denote the trace set obtained from T by evaluating the quantifier $\exists a^{\mathcal{H}}$. →page 35

- The variable $a^{\mathcal{H}}$ is used to encode the countable set $\mathcal{H}$ of sets of traces given by Lemma 3.11. For each $i \in \mathbb{N}$, $a^{\mathcal{H}}$ encodes the set $\{t[0]_{|AP}\, t[1]_{|AP} \ldots \mid t \in T', a^{\mathcal{H}} \in t[i]\} \in \mathcal{H}$, i.e., all traces on which $a^{\mathcal{H}}$ holds in position i. Note that the choice of $a^{\mathcal{H}}$ may depend on T.
- The uniformly quantified variable p points to a set from $\mathcal{H}$. The set indicated by p encodes the set that is used in the current evaluation. We use the formula $\Diamond(p_\pi \wedge a^{\mathcal{H}}_\pi)$ to check if trace π is in the set of traces encoded by p.
- The uniformly quantified variable r refers to the time step that is used for the current evaluation.

After fixing a suitable interpretation of $a^{\mathcal{H}}$, the conjunct $a^{\mathcal{H}}_\pi \wedge p_\pi \wedge r_\pi$ expresses that the current trace set contains all traces of T and that we start the evaluation at point in time i. The translation is defined inductively as follows:

$$\begin{aligned}
[a]^{\mathrm{HQPTL^+}}_{(p,r)} &:= \forall\pi.\Diamond(p_\pi \wedge a^{\mathcal{H}}_\pi) \rightarrow \Diamond(r_\pi \wedge a_\pi)\\
[\neg a]^{\mathrm{HQPTL^+}}_{(p,r)} &:= \forall\pi.\Diamond(p_\pi \wedge a^{\mathcal{H}}_\pi) \rightarrow \Diamond(r_\pi \wedge \neg a_\pi)\\
[\psi_1 \circledvee \psi_2]^{\mathrm{HQPTL^+}}_{(p,r)} &:= [\psi_1]^{\mathrm{HQPTL^+}}_{(p,r)} \vee [\psi_2]^{\mathrm{HQPTL^+}}_{(p,r)}\\
[\psi_1 \wedge \psi_2]^{\mathrm{HQPTL^+}}_{(p,r)} &:= [\psi_1]^{\mathrm{HQPTL^+}}_{(p,r)} \wedge [\psi_2]^{\mathrm{HQPTL^+}}_{(p,r)}\\
[\bigcirc\psi]^{\mathrm{HQPTL^+}}_{(p,r)} &:= \exists r'.\Box(r_\pi \leftrightarrow \bigcirc r'_\pi) \wedge [\psi]^{\mathrm{HQPTL^+}}_{(p,r')}\\
[{\sim}\bot]^{\mathrm{HQPTL^+}}_{(p,r)} &:= \exists\pi.\Diamond(p_\pi \wedge a^{\mathcal{H}}_\pi)\\
[\mathsf{A}^1\psi]^{\mathrm{HQPTL^+}}_{(p,r)} &:= \forall\pi.\Diamond(p_\pi \wedge a^{\mathcal{H}}_\pi) \rightarrow \Diamond(r_\pi \wedge \psi^*_\pi)\\
[\psi_1 \vee \psi_2]^{\mathrm{HQPTL^+}}_{(p,r)} &:= \exists p'.\exists p''.\Diamond(p_\pi \wedge a^{\mathcal{H}}_\pi) \leftrightarrow \Diamond((p'_\pi \vee' p'_\pi) \wedge a^{\mathcal{H}}_\pi)\\
&\quad\ \wedge [\psi_1]^{\mathrm{HQPTL^+}}_{(p',r)} \wedge [\psi_2]^{\mathrm{HQPTL^+}}_{(p'',r)}\\
[\psi_1\,\mathcal{U}\,\psi_2]^{\mathrm{HQPTL^+}}_{(p,r)} &:= \exists_p r'.r \leq r' \wedge [\psi_2]^{\mathrm{HQPTL^+}}_{(p,r')} \wedge \forall_p r''.r \leq r'' \wedge r'' < r' \rightarrow [\psi_1]^{\mathrm{HQPTL^+}}_{(p,r'')}\\
[\psi_1\,\mathcal{W}\,\psi_2]^{\mathrm{HQPTL^+}}_{(p,r)} &:= \forall_p r'.r \leq r' \rightarrow [\psi_1]^{\mathrm{HQPTL^+}}_{(p,r')} \vee \exists_p r''.r \leq r'' \wedge r'' \leq r' \wedge [\psi_2]^{\mathrm{HQPTL^+}}_{(p,r'')}
\end{aligned}$$

where $r < r' := \Box(r_\pi \leftrightarrow \bigcirc\Diamond r'_\pi)$, and $r \leq r' := \Box(r_\pi \rightarrow \Diamond r'_\pi)$. In case of $\mathsf{A}^1\psi$, we use the LTL formula ψ^* from Corollary 3.10. As $\mathsf{A}^1\psi$ is a flat formula, $\mathsf{A}^1\psi$ is equivalent to ψ^* under the standard LTL semantics. For the LTL formula ψ^*, we write ψ^*_π for the equivalent $\forall$ HyperLTL formula that quantifies a single π and annotates every atomic proposition of ψ^* with π. The resulting HyperQPTL$^+$ formula can be transformed to an equivalent prenex formula of the required fragments. □

3.2.3 TeamLTL versus HyperQPTL

In this subsection, we introduce a left-flat fragment for TeamLTL($\circledvee$, A^1) formulas and show that formulas from that fragment can be translated to HyperQPTL.

Definition 3.2 (The left-flat fragment)**.** Let $\mathcal{A}$ be a collection of atoms and operators. A TeamLTL($\mathcal{A}$) formula belongs to the left-flat fragment if in each of its subformulas of the form $\psi_1\,\mathcal{U}\,\psi_2$ or $\psi_1\,\mathcal{W}\,\psi_2$, ψ_1 is a flat formula.

Such defined fragment allows for arbitrary use of the $\Diamond$ operator, and therefore remains incomparable to HyperLTL [153]. For instance,

$$\Diamond\,\mathrm{dep}(a,b) \vee \Diamond\,\mathrm{dep}(c,d)$$

is a nontrivial formula in this fragment. It states that the set of traces can be partitioned into two parts, one where eventually a determines the value of b, and another one where eventually c determines the value of d. The property is not expressible in HyperLTL, because HyperLTL cannot state the property "there is a point in time at which p holds on all (or infinitely many) traces" [34]. The relation of left-flat TeamLTL($\circledcirc$, A^1) to FO$[<, E]$ remains open.

Left-flatness is a semantic property. We show that checking a TeamLTL($\mathcal{A}$) formula for flatness of is strongly tied to the satisfiability problem of the logic. If $\mathsf{A}^1 \in \mathcal{A}$, then the problems are even equivalent.

Lemma 3.13. *Let $\mathcal{A} \subseteq \{\subseteq, \circledcirc, {\sim}\bot, \mathsf{A}^1\}$. If satisfiability checking of TeamLTL($\mathcal{A}, \mathsf{A}^1$) is decidable, then so is checking flatness of TeamLTL($\mathcal{A}, \mathsf{A}^1$) formulas. If satisfiability of TeamLTL($\mathcal{A}$) is undecidable, then so is checking flatness of TeamLTL($\mathcal{A}$) formulas.*

Proof. For the first direction, assume that the satisfiability problem of TeamLTL($\mathcal{A}, \mathsf{A}^1$) is decidable. Then a TeamLTL($\mathcal{A}$) formula ψ is flat iff $\psi \nleftrightarrow \mathsf{A}^1\psi$ is unsatisfiable. For the other direction, we show that for any formula $\psi \in$ TeamLTL($\mathcal{A}$), ψ is unsatisfiable iff ψ is flat and ψ^* from Corollary 3.10 is not satisfiable under the LTL semantics. Again, we prove the equivalence by distinguishing two direction. First, assume that ψ is unsatisfiable. Since no trace set T satisfies ψ, it is trivially flat. By Corollary 3.10, ψ^* is not satisfiable under the LTL semantics. Conversely, suppose ψ is flat and ψ^* is not satisfiable. By Corollary 3.10, $\{t\}, i \not\models \psi$ for every t. It then follows, by flatness, that $T, i \not\models \psi$ for every team T. Thus ψ is not satisfiable. Now, since LTL satisfiability is in PSPACE [211], we can conclude that checking flatness of ψ must be undecidable, otherwise unsatisfiability checking of ψ would be decidable. □

There exist only few results on the satisfiability problem of TeamLTL and its extensions. Satisfiability of TeamLTL without additional connectives is reducible to LTL satisfiability checking [153]. We expect that this result carries over to TeamLTL(A^1). Satisfiability checking for TeamLTL($\subseteq, \circledcirc$) is undecidable [225]. As a remedy, the A^1 operator defines a syntactic fragment that guarantees flatness. This fragment is well-defined in the sense that, by definition, a TeamLTL formula ψ is flat iff it is equivalent to the formula $\mathsf{A}^1\psi$. In the following, we therefore assume that for all subformulas of the form $\psi_1 \mathcal{U} \psi_2$ and $\psi_1 \mathcal{W} \psi_2$, ψ_1 is of the form $\mathsf{A}^1\psi_1'$.

We now describe a translation from the left-flat fragment of TeamLTL($\circledcirc$, A^1) to the $\exists_p^*\forall_\pi$ fragment of HyperQPTL. We make use of the fact that satisfaction of flat formulae ψ can be determined with the usual (single-traced) LTL semantics. In the evaluation of ψ, it is thus sufficient to consider only finitely many trace subsets, whose temporal behavior can be reflected by existentially quantified p-sequences.

Theorem 3.14. *$\exists_p^*\forall_\pi$ HyperQPTL subsumes the left-flat fragment of TeamLTL($\circledcirc$, A^1).*

Proof. We describe a linear compositional translation $[\cdot]_r^{\mathrm{HQPTL}}$ such that the following equivalence holds for every trace set T.

$$T \models \psi \quad \text{iff} \quad T \models \exists r^0, r^{\psi_1}, \dots, r^{\psi_n}, d^{\rho_1}, \dots, d^{\rho_m}. \forall \pi. r^0 \wedge \bigcirc \Box \neg r^0 \wedge [\psi]_{r^0}^{\mathrm{HQPTL}}$$

Above, $[\psi]_{r^0}^{\mathrm{HQPTL}}$ is a quantifier-free formula with free propositional variables r^0, and $r^{\psi_1}, \dots, r^{\psi_n}$ and $d^{\rho_1}, \dots, d^{\rho_m}$, and a free trace variable π. The idea of the translation is that each propositional variable r^{ψ_i} indicates a point in time at which a subformula ψ_i of ψ must hold true. Propositional variables d^{ρ_i} resolve the decision of $\circledvee$-choices for subformula ρ_i. The key observation is that the creation of subteams with $\vee$ is only possible in positive positions of the formula. In other words, all $\vee$ operators are in the scope of existential temporal decisions. Thus, a team is described by the existentially chosen points at which all traces have to synchronize. As the split is also an existential decision, and there are only finitely many synchronization points (the bound is given by the number of subformulas in ψ), there are only finitely many teams that need to be considered in the evaluation. The synchronization points are encoded with the propositional variables, and the universal quantifier $\forall \pi$ sorts each trace into one of the finitely many teams. We define $[\cdot]_r^{\mathrm{HQPTL}}$ as follows.

$$\begin{array}{lcl}
[p]_r^{\mathrm{HQPTL}} & := & \Box(r \rightarrow p_\pi) \\
[\neg p]_r^{\mathrm{HQPTL}} & := & \Box(r \rightarrow \neg p_\pi) \\
[\bigcirc \psi]_r^{\mathrm{HQPTL}} & := & \Box(r \leftrightarrow \bigcirc r^{\psi}) \wedge [\psi]_{r^\psi}^{\mathrm{HQPTL}} \\
[\mathsf{A}^1 \psi]_r^{\mathrm{HQPTL}} & := & \Box(r \rightarrow \psi^*) \\
[\psi_1 \wedge \psi_2]_r^{\mathrm{HQPTL}} & := & [\psi_1]_r^{\mathrm{HQPTL}} \wedge [\psi_2]_r^{\mathrm{HQPTL}} \\
[\psi_1 \vee \psi_2]_r^{\mathrm{HQPTL}} & := & [\psi_1]_r^{\mathrm{HQPTL}} \vee [\psi_2]_r^{\mathrm{HQPTL}} \\
[\psi_1 \circledvee \psi_2]_r^{\mathrm{HQPTL}} & := & (d^{\psi_1 \circledvee \psi_2} \rightarrow [\psi_1]_r^{\mathrm{HQPTL}}) \wedge (\neg d^{\psi_1 \circledvee \psi_2} \rightarrow [\psi_2]_r^{\mathrm{HQPTL}}) \\
[\psi_1 \mathcal{U} \psi_2]_r^{\mathrm{HQPTL}} & := & \Box(r \rightarrow r^{\psi_1} \mathcal{U} r^{\psi_2}) \wedge \Box(r^{\psi_1} \rightarrow \psi_1^*) \wedge [\psi_2]_{r^{\psi_2}}^{\mathrm{HQPTL}} \\
[\psi_1 \mathcal{W} \psi_2]_r^{\mathrm{HQPTL}} & := & \Box(r \rightarrow r^{\psi_1} \mathcal{W} r^{\psi_2}) \wedge \Box(r^{\psi_1} \rightarrow \psi_1^*) \wedge [\psi_2]_{r^{\psi_2}}^{\mathrm{HQPTL}}
\end{array}$$

In the above translation, ψ^* and ψ_1^* are the corresponding LTL formulas from Corollary 3.10. We assume that all propositional variables occurring in the translation are quantified as described above. □

3.3 The Hierarchy of Branching-time Hyperlogics

Similar to the case for linear-time logics, CTL* is expressively equivalent to MPL, and the extension of CTL* with propositional quantification, QCTL*, is equivalent to MSO (see Figure 3.1b). We show that the observations from the linear-time hierarchy of hyperlogics mostly also apply to the branching-time hierarchy: the equal-level predicate adds in general more expressiveness to FO/SO logics than path quantifiers add to temporal logics.

There is one difference, though. According to the definition of QCTL*, propositional quantifiers may choose the value of the quantified proposition differently in every node of the tree. We show that the straight-forward extension to hyperproperties, HyperQCTL*, has the same expressiveness as MSO extended with the equal-level predicate. This observation strongly resembles the observation in the linear-time hierarchy, where HyperQPTL⁺ has the expressiveness of S1S[E]. Inspired by HyperQPTL, we therefore introduce HyperQ^{-}CTL* as a weaker version of HyperQCTL*. And indeed, the expressiveness of HyperQ^{-}CTL* falls in between the ones of MPL[E] and HyperQCTL*.

MPL[E]. MPL[E] is a straight-forward extension of →MPL with the equal-level predicate. We just add the additional term $E(x, y)$ to the syntax of MPL. The semantics of the operator is defined as follows.

→page 26

$$\mathcal{V}_1, \mathcal{V}_2 \models_{(\mathcal{T},L)} E(x, y) \quad \text{iff} \quad |\mathcal{V}_1(x)| = |\mathcal{V}_1(y)|$$

Lemma 3.15. *MPL[E] is at least as expressive as HyperCTL*.*

Proof. We use MPL's first-order quantification to keep track of the current time and use its second-order quantification to quantify the correct path. Given a HyperCTL* formula φ, and a first-order variable i, we inductively construct an MPL[E] formula $[\varphi]_i^{\mathrm{MPL}[E]}$ as follows.

$$\begin{aligned}
[a_\pi]_i^{\mathrm{MPL}[E]} &:= \exists x.\, x \in X_\pi \land E(x, i) \land x \in X_a \\
[\neg\varphi]_i^{\mathrm{MPL}[E]} &:= \neg[\varphi]_i^{\mathrm{MPL}[E]} \\
[\varphi_1 \lor \varphi_2]_i^{\mathrm{MPL}[E]} &:= [\varphi_1]_i^{\mathrm{MPL}[E]} \lor [\varphi_2]_i^{\mathrm{MPL}[E]} \\
[\mathsf{X}\,\varphi]_i^{\mathrm{MPL}[E]} &:= \exists j > i.\, \neg(\exists k.\, i < k < j) \land [\varphi]_j^{\mathrm{MPL}[E]} \\
[\varphi_1 \,\mathsf{U}\, \varphi_2]_i^{\mathrm{MPL}[E]} &:= \exists j \geq i.\, [\varphi_2]_j^{\mathrm{MPL}[E]} \land \forall k.\, i \leq k < j \rightarrow [\varphi_1]_k^{\mathrm{MPL}[E]} \\
[\exists\pi.\, \varphi]_i^{\mathrm{MPL}[E]} &:= \exists X_\pi.\, \mathit{prefix}(X_\pi, X_\varepsilon, i) \land [\varphi]_i^{\mathrm{MPL}[E]}
\end{aligned}$$

As MPL's second-order quantification always spans full (infinite) paths, we only need to guarantee that the newly quantified path branches of the current one at position i. This is encoded in the *prefix* function defined as

$$\mathit{prefix}(X_\pi, X_\varepsilon, i) := \forall j \leq i, x.\, E(x, j) \rightarrow x \in X_\pi \leftrightarrow x \in X_\varepsilon$$

where we use X_ε to denote the second-order variable that was quantified most recently (i.e. closest in the scope to X_π). X_ε can be easily obtained from the syntax tree of the formula. If X_π is not in the scope of any other second-order quantifier (i.e., it is the first path that is quantified), then we replace $\mathit{prefix}(X_\pi, X_\varepsilon, i)$ with *true*. Finally, for any HyperCTL* formula φ, we obtain the following equivalent MPL[E] formula.

$$[\varphi]^{\mathrm{MPL}[E]} := \exists z.\, \forall z'.\, \neg z' < z \land [\varphi]_z^{\mathrm{MPL}[E]}$$

The formula quantifies the root level of the tree and evaluates the formula at this point. □

Theorem 3.16. *MPL[E] is strictly more expressive than HyperCTL*.*

Proof. It is known that HyperCTL* cannot express the property “there exists a point in time at which all traces agree on the value of a” [34]. In MPL[E], we can express the property as follows.

$$\exists i.\, \forall x, y.\, E(x, i) \land E(y, i) \rightarrow x \in X_a \leftrightarrow y \in X_a$$

With Lemma 3.15, the statement follows. □

Variants of HyperQCTL*. We define two extensions of HyperCTL* with propositional quantification: Hyper$\overline{\text{Q}}$CTL* and HyperQCTL*. The more straight-forward extension of QCTL* with path quantifiers (similar to how HyperCTL* extends CTL*) yields HyperQCTL*. The less expressive variant, HyperQCTL*, requires that the quantified propositional variable has the same value on nodes which reside on the same level of the tree. Syntactically, both logics add propositional quantification to the syntax of HyperCTL*. As for HyperQPTL, we define the syntax of Hyper$\overline{\text{Q}}$CTL* to additionally contain a rule p for an atomic proposition without path variable. Note that in both logics, both types of quantifiers (propositional and path) are allowed to occur in the scope of temporal operators. The semantics of propositional quantification in HyperQCTL* is defined as follows.

$$\Pi, i \models_{(\mathcal{T},L)} \exists p.\, \varphi \quad \text{iff} \quad \exists L' : S \rightarrow 2^{AP}.\, L' =_{AP\setminus\{p\}} L \text{ and } \Pi, i \models_{(\mathcal{T},L')} \varphi$$

For Hyper$\overline{\text{Q}}$CTL*, the semantics requires the new labeling function to assign p uniformly across all paths. We also need to give an evaluation for the propositional variable without the path annotation.

$$\begin{aligned}
\Pi, i \models_{(\mathcal{T},L)} p \quad & \text{iff} \quad \forall r \in \mathit{finPaths}(\mathcal{T}).\, |r| = i + 1 \rightarrow p \in L(r[i]) \\
\Pi, i \models_{(\mathcal{T},L)} \exists p.\, \varphi \quad & \text{iff} \quad \exists t_p \in (2^{\{p\}})^\omega, \exists L' : S \rightarrow 2^{AP}.\, L' =_{AP\setminus\{p\}} L \text{ and} \\
& \qquad \forall r \in \mathit{paths}(\mathcal{T}).\, \mathit{trace}_{L'}(r) =_{\{p\}} t_p \text{ and } \Pi, i \models_{(\mathcal{T},L')} \varphi
\end{aligned}$$

In the following, if we need to distinguish between the type of quantification, we use $\exists p.\, \varphi$ for Hyper$\overline{\mathrm{Q}}$CTL* and $\exists a.\, \varphi$ for HyperQCTL*. Similarly to the case of linear-time hyperlogics, we obtain the result that Hyper$\overline{\mathrm{Q}}$CTL*, even though it is restricted to uniform quantification, is strictly more expressive than MPL[E].

Lemma 3.17. *Hyper$\overline{\mathrm{Q}}$CTL* subsumes MPL[E].*

Proof. The proof is very similar to the proof that HyperQPTL subsumes FO[$<, E$] given in Lemma 3.1. As before, we encode MPL[E]'s first-order quantification with a combination of path quantification and propositional quantification to encode the time. The resulting formula has no quantifier in the scope of a temporal operator. It is therefore, syntactically, a HyperQPTL formula (interpreted on a tree). We only give the rules in which the translation differs from the translation $[\cdot]^{\mathrm{HQPTL}}$ given in Lemma 3.1. As MPL[E]'s second-order quantification quantifies full paths, we just use a path quantifier to encode the set.

$$\begin{aligned}
[x \in X_a]^{\mathrm{H}\overline{\mathrm{Q}}\mathrm{CTL}^*} &:= \mathrm{G}(p^x \rightarrow a_{\pi^x}) \\
[x \in X]^{\mathrm{H}\overline{\mathrm{Q}}\mathrm{CTL}^*} &:= \mathit{equalTrace}(\pi^X, \pi^x) \\
[\exists X.\, \varphi]^{\mathrm{H}\overline{\mathrm{Q}}\mathrm{CTL}^*} &:= \exists \pi^X.\, [\varphi]^{\mathrm{H}\overline{\mathrm{Q}}\mathrm{CTL}^*}
\end{aligned}$$

Above, X_a is a set that is free in the MPL[E] formula (i.e., it is interpreted by the tree the formula is evaluated on), whereas X is quantified by a second-order quantifier. □

Theorem 3.18. *Hyper$\overline{\mathrm{Q}}$CTL* is strictly more expressive than MPL[E].*

Proof. With Lemma 3.17, we are left to show that Hyper$\overline{\mathrm{Q}}$CTL* is strictly more expressive than MPL[E]. The proof proceeds as the one for Theorem 3.2. Consider the model of *linear trees*, i.e., trees in which each node has a unique successor. For this class of models, MPL[E] is equivalent to FO[$<$], since the equal-level predicate collapses to equality and second-order quantification in MPL[E] can only quantify the unique single path. Likewise, Hyper$\overline{\mathrm{Q}}$CTL* collapses to QPTL. But it is known that QPTL can express all ω-regular properties [212] and FO[$<$], which is equivalent to LTL, cannot [178]. □

Finally, the unrestricted propositional quantification is more expressive than uniform propositional quantification, as in the case of linear-time hyperlogics.

Theorem 3.19. *HyperQCTL* is strictly more expressive than Hyper$\overline{\mathrm{Q}}$CTL*.*

Proof. HyperQCTL* subsumes Hyper$\overline{\mathrm{Q}}$CTL*, as we can state with two universal quantifiers that a proposition p has to be chosen uniformly for all traces: $\forall \pi, \pi'.\, \mathrm{G}(p_\pi \leftrightarrow p_{\pi'})$. To show that HyperQCTL* is strictly more expressive, consider the class of models in which every node in the tree has exactly one successor except for the root, which may have infinitely many successors. On this class of models, HyperQCTL* is equivalent to HyperQPTL$^+$ and HyperQ$^-$CTL* collapses to HyperQPTL. As HyperQPTL$^+$ is strictly more expressive than HyperQPTL by Theorem 3.7, the statement follows. □

Lastly, we extend MSO with the equal-level predicate. As for MPL[E], the extension is straight-forward in the branching-time setting.

MSO[E]. MSO[E] adds the equal level predicate (as defined for MPL[E] at the beginning of this section) to the definition of →MSO on trees. It thus allows for full second-order quantification. →page 27

Theorem 3.20. *HyperQCTL* and MSO[E] are expressively equivalent.*

Proof. To prove that HyperQCTL* subsumes MSO[E], we build on the translation from MPL[E] to HyperQ$^-$CTL* in the proof of Lemma 3.17 and alter the rules for the second-order quantification.

$$\begin{aligned}[x \in X]^{\mathrm{HQCTL}^*} &= \mathrm{G}(a^x_{\pi^x} \rightarrow a^X_{\pi^x}) \\ [\exists X.\, \varphi]^{\mathrm{HQCTL}^*} &= \exists a^X.\, [\varphi]^{\mathrm{HQCTL}^*}\end{aligned}$$

Conversely, we build on the translation from HyperCTL* to MPL[E] from Lemma 3.15. The translation uses the fact that MPL[E]'s second-order quantification can only quantify full paths. This we have to encode in MSO[E]. We define a predicate $\mathit{fullPath}(X)$ that expresses that a set X contains exactly the nodes of one path.

$$\mathit{fullPath}(X) := (\exists x.\, \forall y.\, y \not< x) \wedge \forall x \in X.\, (\exists y \in X.\, y > x) \wedge (\forall z \in X.\, E(x,z) \rightarrow x = z)$$

The formula expresses that X contains the root of the tree, that every element contains at least one successor, and that there are no two different nodes on the same level. For the translation from HyperQCTL* to MSO[E], we only give the rules that differ from the translation from HyperCTL* to MPL[E] in Lemma 3.15.

$$\begin{aligned}[\exists \pi.\, \varphi]_i^{\mathrm{MSO}[E]} &= \exists X_\pi.\, \mathit{fullPath}(X_\pi) \wedge \mathit{prefix}(X_\pi, X_\varepsilon, i) \wedge [\varphi]_i^{\mathrm{MSO}[E]} \\ [\exists a.\, \varphi]_i^{\mathrm{MSO}[E]} &= \exists X_a.\, [\varphi]_i^{\mathrm{MSO}[E]}\end{aligned}$$

As in the proof before, the final formula first quantifies the root level of the tree. □

Chapter 4

Satisfiability of Temporal Safety and Temporal Liveness

The satisfiability problem is a key question to understand a logic. Algorithmically, it constitutes an important preprocessing step (e.g., to detect implications between specifications) and has a long tradition as target in reductions (see →Section 1.3 for an introduction). →page 8 Reasoning about satisfiability for hyperproperties is significantly harder than it is for trace properties. LTL satisfiability checking is PSPACE-complete [211]. For HyperLTL, the problem is highly undecidable in general, namely Σ_1^1-complete [98]. While formulas from the $\exists^*\forall^*$ fragment can still be decided in EXPSPACE [82], $\forall\exists$ quantifier alternations quickly lead to undecidability. Nevertheless, the $\forall^*\exists^*$ fragment contains many relevant properties like generalized noninterference [176], program refinement [51], and software doping [62]. Despite its importance, positive results for the $\forall^*\exists^*$ fragment have been very rare and were only obtained by heavy restrictions on the use of temporal operators or by assuming finite models [173] (see related work in →Chapter 5). Algorithms, even if incomplete, are similarly missing. →page 71

In this chapter, we aim to define expressive fragments of HyperLTL that comprise formulas with $\forall\exists$ quantifier alternations but also have a simpler satisfiability problem. Towards this goal, we define the notion of *temporal safety* and *temporal liveness*: a HyperLTL formula is *temporal safety* (resp. *temporal liveness*) if its LTL body describes a safety (resp. liveness) property. We first show that for studying satisfiability, our fragments are more suitable than the existing definition of *hypersafety* and *hyperliveness* defined by Clarkson and Schneider [51]. The classification into safety and liveness properties has a long tradition in the study of trace properties, where especially the restriction to safety properties often yields easier algorithms. Indeed, we show that for the temporal safety fragment, the complexity of the problem drops to coRE-completeness. We obtain the result by a reduction to first-order logic, which opens the door for the use of common first-order techniques such as resolution, tableau, and related methods [198] for the reasoning with hyperproperties. For temporal liveness, on the other hand, we show that any HyperLTL formula can be translated to an equisatisfiable formula in the temporal liveness fragment. This shows that the fragment stays Σ_1^1-complete.

Finally, to complement our results, we propose a general approximation algorithm to find the *largest* model of $\forall\exists^*$ HyperLTL specifications. Our experimental evaluation shows that our algorithm nicely complements the only other existing approach that can handle $\forall\exists^*$ formulas, which iteratively searches for models of bounded size [83]. Additionally, out implementation can also prove unsatisfiability for some formulas (which is impossible in bounded approaches).

Outline. In Section 4.1, we define the notion of temporal safety, compare the definition to hypersafety, and present our complexity results for temporal safety hyperproperties. Section 4.2 has the same structure but for liveness properties. Finally, Section 4.3 proposes the new algorithm for finding largest models of $\forall\exists^*$ HyperLTL formulas including an evaluation of the algorithm in comparison to existing tools.

Publications. This chapter is based on the following publication.

[25] Raven Beutner, David Carral, Bernd Finkbeiner, Jana Hofmann, and Markus Krötzsch. **Deciding Hyperproperties Combined with Functional Specifications**. *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2022).*

4.1 Temporal Safety

In this section, we study the satisfiability problem of temporal safety HyperLTL formulas. We begin by defining temporal safety and argue why, compared with hypersafety, it is the more suitable fragment in the context of satisfiability. Subsequently, we show that temporal safety specifications reduce the general Σ_1^1-hardness of HyperLTL [98] to coRE-complete. Even though we are primarily interested in simplifying the satisfiability problem for $\forall^*\exists^*$ formulas, we state definitions and theorems more generally for full HyperLTL if possible.

The following two propositions establish that the fragment of →Definition 2.4 formulas does not really simplify the satisfiability problem of HyperLTL. The main reason is that deciding if a HyperLTL formula describes a hypersafety property is already highly undecidable. This follows from the fact that deciding whether a formula is hypersafety can be encoded in unsatisfiability of HyperLTL [81] and that HyperLTL unsatisfiability checking is Π_1^1-hard [98]. →page 20

Proposition 4.1. *Deciding if a HyperLTL formula is hypersafety is* Π_1^1*-hard.*

On the other hand, if we know that a HyperLTL formula describes a hypersafety property, then deciding satisfiability is not harder than deciding LTL satisfiability.

Proposition 4.2. *Given a HyperLTL formula φ that describes a hypersafety property, satisfiability of φ is decidable in PSPACE.*

Proof. As hypersafety properties are closed under subsets [51], φ is satisfiable iff it is satisfiable by a single trace model. Therefore, we can change all quantifiers in φ to universal ones, resulting in an equisatisfiable (but not equivalent) $\forall^*$ formula, for which satisfiability is decidable in PSPACE [82]. □

To obtain a fragment that simplifies the satisfiability problem and for which which membership is still decidable, we introduce the fragment of temporally safe formulas.

Definition 4.1. A HyperLTL formula $Q\pi_1 \dots Q\pi_n. \psi$ is a *temporal safety* formula if ψ (interpreted as an LTL formula over $AP_{\pi_1} \cup \dots \cup AP_{\pi_n}$) describes a safety property.

As safety is recognizable for LTL [210], membership of temporal safety is decidable as well. We show that temporal safety also constitutes a larger subclass of properties than hypersafety (within the $\forall^*\exists^*$ fragment). Hypersafety properties are closed under subsets and are therefore always satisfiable by a single trace model (if satisfiable at all) [51]. Temporal safety, on the other hand, can enforce models with infinitely many traces as witnessed by the following formula.

$$(\exists \pi. a_\pi) \land (\forall \pi. \square(a_\pi \rightarrow \bigcirc\square\neg a_\pi)) \land (\forall \pi. \exists \pi'. \square(a_\pi \leftrightarrow \bigcirc a_{\pi'}))$$

The formula above can be arranged as a $\forall^*\exists^*$ HyperLTL formula. To satisfy the formula, proposition a may only hold once per trace. Additionally, for every trace, there exists one where a occurs one step later. We now show that $\forall^*$ temporal safety subsumes all $\forall^*\exists^*$ hypersafety formulas.

Lemma 4.3. *For every $\forall^*\exists^*$ formula describing a hypersafety property, there exists an equivalent $\forall^*$ temporal safety formula.*

We prove the above lemma with the following two propositions. The first proposition establishes that $\forall^*\exists^*$ hypersafety formulas can be equivalently expressed with universal formulas.

Proposition 4.4. *For every $\forall^*\exists^*$ formula describing a hypersafety property, there exists an equivalent $\forall^*$ formula.*

Proof. Let $\varphi = \forall \pi_1 \dots \pi_n. \exists \pi'_1 \dots \pi'_m. \psi$ be a hypersafety formula. For any function $g : \{1, \dots, m\} \rightarrow \{1, \dots, n\}$ (of which there are n^m many), we define $\psi_{[g]}$ as the formula obtained by replacing each trace variable π'_i with $\pi_{g(i)}$. Now define

$$\varphi' := \forall \pi_1 \ldots \pi_n. \bigvee_{g:\{1,\ldots,m\}\to\{1,\ldots,n\}} \psi_{[g]}$$

We claim that $\varphi \equiv \varphi'$. It is easy to see that φ' implies φ, as the disjunction gives an explicit witness for the existential quantifiers. For the other direction, assume $T \models \varphi$ for some model T. Let $t_1, \ldots, t_n \in T$ be arbitrary. As φ is a hypersafety property (cf. →Definition 2.4) and $\{t_1, \ldots, t_n\} \subseteq T$, we get that $\{t_1, \ldots, t_n\} \models \varphi$. In particular, if we bind each π_i to t_i (in φ), we get witness traces $t'_1, \ldots, t'_m \in \{t_1, \ldots, t_n\}$ for the existential quantifiers in φ. Now define g by mapping each $1 \leq j \leq m$ to $i \in \{1, \ldots, n\}$ such that $t'_j = t_i$. The trace assignment $[\pi_1 \mapsto t_1, \ldots, \pi_n \mapsto t_n]$ satisfies $\psi_{[g]}$. As we can find such a g for every $t_1, \ldots, t_n \in T$, we get that $T \models \varphi'$ as required. □ →page 20

Note that by definition of hypersafety, if φ is hypersafety and $\varphi' \equiv \varphi$, then φ' is hypersafety. Thus, the following proposition finishes the proof of Lemma 4.3.

Proposition 4.5. *For every $\forall^*$ hypersafety formula, there exists an equivalent $\forall^*$ temporal safety formula.*

Proof. Let $\varphi = \forall \pi_1 \ldots \pi_n. \psi$ be a hypersafety HyperLTL formula. Similarly to the proof of Proposition 4.4, for any function $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$, we define the formula $\psi_{[f]}$ as the formula obtained by replacing each trace variable π_i for $1 \leq i \leq n$ with $\pi_{f(i)}$. We define $\varphi' := \forall \pi_1 \ldots \pi_n. \psi'$ where

$$\psi' := \bigwedge_{f:\{1,\ldots,n\}\to\{1,\ldots,n\}} \psi_{[f]}$$

By the semantics of universal quantification, it is easy to see that $\varphi \equiv \varphi'$. We claim that ψ' expresses a safety property when interpreted as trace property over $AP_{\pi_1} \cup \ldots \cup AP_{\pi_n}$. Take any trace t over $AP_{\pi_1} \cup \ldots \cup AP_{\pi_n}$ with $t \not\models \psi'$ (as in the definition of safety, cf. →Definition 2.2). We need to show that there exists a $u \sqsubseteq t$ such that for any t' with $u \sqsubseteq t'$, $t' \not\models \psi'$. Let $T = \{t_1, \ldots, t_n\}$ be the set obtained by splitting t into n traces, i.e., t_i is a trace over AP that mirrors the assignments of propositions in AP_{π_i} in t. By construction of T, we get $T \not\models \varphi'$ and, as $\varphi \equiv \varphi'$ and φ is hypersafety, we get a finite set of finite traces $U \sqsubseteq T$ such that no extension of U satisfies φ. We assume that w.l.o.g. that $U = \{u_1, \ldots, u_n\}$, where $u_i \sqsubseteq t_i$ for each i. This assumption is valid, as we can replace multiple prefixes of the same t_i by the longest among those prefixes. Furthermore, we can add an arbitrary prefix of each t_i that previously had no prefix in U and maintain the property that any extension of U does not satisfy φ. We further assume, again w.l.o.g., that all u_i have the same length, say k. Now define u as the finite trace (of length k) over $AP_{\pi_1} \cup \ldots \cup AP_{\pi_n}$, where the assignment to →page 19

AP_{π_i} is taken from u_i. As $u_i \sqsubseteq t_i$ for each i, we get $u \sqsubseteq t$. It remains to argue that u cannot be extended to a trace t' such that $t' \models \psi'$. Let t' be any trace with $u \sqsubseteq t'$. Again, we split t' into traces $t'_1, \ldots, t'_n$. Now, $T' = \{t'_1, \ldots, t'_n\}$ satisfies $U \sqsubseteq T'$, so $T' \not\models \varphi$. By the semantics of universal quantification, there thus exists an f such that $[\pi_1 \mapsto t'_{f(1)}, \ldots, \pi_n \mapsto t'_{f(n)}] \not\models \psi$ and therefore $[\pi_1 \mapsto t'_1, \ldots, \pi_n \mapsto t'_n] \not\models \psi_{[f]}$. This implies that $t' \not\models \psi_{[f]}$ in the LTL semantics and we obtain $t' \not\models \psi'$ as required. □

We do not claim that every $\forall^*$ hypersafety formula is temporally safe. Instead, Proposition 4.5 only states that there exists an equivalent temporally safe formula. For example, $\forall\pi\forall\pi'. \Diamond(a_\pi \land \neg a_{\pi'})$ is unsatisfiable and thus a hypersafety property but $\Diamond(a_\pi \land \neg a_{\pi'})$ is not a safety property.

Having established that temporal safety spans a broad spectrum of properties, we now establish that the general analytical hardness of HyperLTL satisfiability checking [98] drops to coRE-completeness for temporal safety. We show the upper bound by giving an effective translation from temporally safe HyperLTL to first-order logic using the fact that satisfiability of first-order logic is coRE-complete [115]. Our translation thus enables the application of first-order satisfiability solvers in the realm of hyperproperties.

Lemma 4.6. *The satisfiability problem of temporally safe HyperLTL is in coRE.*

Proof. Let $\varphi = Q_1\pi_1 \ldots Q_n\pi_n. \psi$ be a temporally safe HyperLTL formula. Let $\mathcal{A}_\psi = (Q_\psi, q_{0,\psi}, \Sigma_\psi, \delta_\psi)$ be a →safety automaton over $\Sigma_\psi = 2^{AP_{\pi_1} \cup \ldots \cup AP_{\pi_n}}$ that accepts ψ (when interpreted as an LTL formula over $AP_{\pi_1} \cup \ldots \cup AP_{\pi_n}$). We define in the following an equisatisfiable first-order formula Θ, which can be computed from φ. For readability, we use two-sorted first-order logic, which is equisatisfiable to standard first-order logic. We use two sorts: *Trace*, which contains trace variables, and *TimePoint*, which contains time variables. We use the constant i_0 : *TimePoint* to indicate the initial time point. The predicate $Succ(\cdot,\cdot)$ over *TimePoint* × *TimePoint* encodes the successor relation on time. For each $a \in AP$, we use a predicate $P_a(\cdot,\cdot)$ over *Trace* × *TimePoint* to indicate that on trace t, a holds at point in time i. Intuitively, Θ encodes the set of accepting runs through $\mathcal{A}$ for the sort *Trace*. For each state $q \in Q_\psi$, we use a predicate $State_q$ over $Trace^n \times TimePoint$. Informally, $State_q(x_1, \ldots, x_n, i)$ indicates that a run of $\mathcal{A}$ on traces $x_1, \ldots, x_n$ is in state q at timepoint i. We first ensure that each point in time has a successor and that the set of traces is non-empty.

→page 19

$$\begin{aligned} \psi_{\text{succ}} &:= \forall i : \mathit{TimePoint}. \exists i' : \mathit{TimePoint}. \mathit{Succ}(i, i') \\ \psi_{\text{non-empty}} &:= \exists x : \mathit{Trace}. \mathit{true} \end{aligned}$$

For each state $q \in Q_\psi$, we construct a formula ρ_q (over free variables $x_1, \ldots, x_n$), describing the next state in an accepting run through $\mathcal{A}$ for traces $x_1, \ldots, x_n$.

$$\rho_q := \forall i, i' : \mathit{TimePoint}.\ \mathit{State}_q(x_1, \ldots, x_n, i) \land \mathit{Succ}(i, i') \\ \rightarrow \bigvee_{(q,A,q') \in \delta_\psi} \Big(\bigwedge_{a_{\pi_j} \in A} P_a(x_j, i) \land \bigwedge_{a_{\pi_j} \notin A} \neg P_a(x_j, i)\ \land \mathit{State}_{q'}(x_1, \ldots, x_n, i') \Big)$$

The formula states that if the run for $x_1, \ldots, x_n$ is at point in time i in state q, and if i' is a successor of i, then the run is in a valid successor state in the next time point.

Note that we cannot encode that for every i, there is a *unique* i' such that $\mathit{Succ}(i, i')$, as this would require an interpreted equality relation. This is not a problem, however, as we only need to enforce the existence of at least one accepting run through the automaton. Similarly, there might be several State_q predicates which are true for $(x_1, \ldots, x_n, i)$. But then, there must be a valid successor state in $\mathcal{A}$ for each of the states $(x_1, \ldots, x_n)$ might be in at point in time i. Finally, Θ is defined as follows

$$\Theta := Q_1 x_1 : \mathit{Trace}. \ldots Q_n x_n : \mathit{Trace}.\ \psi_{\text{succ}} \land \psi_{\text{non-empty}} \land \mathit{State}_{q_0}(x_1, \ldots, x_n, i_o) \land \bigwedge_{q \in Q} \rho_q$$

The last two conjuncts together ensure that all trace tuples chosen by the quantifiers have an infinite run in $\mathcal{A}$ starting in the initial state and in the initial time point. We show that the FOL formula Θ and HyperLTL formula φ are equisatisfiable.

For the first direction, assume Θ is satisfiable and fix a first-order model. The model fixes a domain for the two sorts and the valuation of the predicates on these domains. Let the set X be the set of elements from the *Trace* domain that are assigned to some variable x_i in any possible evaluation of the quantifiers. We iteratively construct a trace for any element of X. To do so, let $i_0, i_1, \ldots$ be a fixed sequence of elements from *TimePoint* such that $\mathit{Succ}(i_j, i_{j+1})$ for any $j \in \mathbb{N}$ and i_0 is the constant described above. This sequence exists by the construction of Θ but might not be unique and elements might occur several times. For each element $v \in X$, we define a trace $t_v \in \Sigma^\omega$, by setting $t_v[n] := \{a \mid P_a(v, i_n)\}$ for every $n \in \mathbb{N}$. We choose $T := \{t_v \mid v \in X\}$ is a model of φ. Let $(v^1, \ldots, v^n)$ be the elements from X assigned to the quantifiers in Θ. We mimic the choice in φ's quantifiers and receive $(t_v^1, \ldots, t_v^n)$. By the construction of ρ_q, $\mathcal{A}_\psi$ has a run on $(t_v^1, \ldots, t_v^n)$, which means that $(t_v^1, \ldots, t_v^n) \models \psi$.

For the other direction, assume that φ is satisfiable by trace set T, which we choose as domain for sort *Trace*. For *TimePoints*, we choose the set of natural numbers with $i_0 = 0$. We set $P_a(t, i)$ to true iff $a \in t[i]$. For every assignment of $(t_1, \ldots t_n)$ to the trace variables, we fix a run through $\mathcal{A}$ and set $\mathit{State}_q(t_1, \ldots, t_n, i)$ to true iff the the run is in state q in step i. The resulting structure satisfies Θ.

Finally, note that our construction is limited to safety automata. For Büchi automata, we could not ensure infinitely many visits to an accepting state. □

To complement the upper bound, we show coRE-hardness by reducing the complement of the halting problem of deterministic Turing machines. With the proof we also show that already a *single* $\square$ operator with nested $\bigcirc$ operators suffices for coRE-hardness. This fits into the line of results from recent years that hardness of a class of HyperLTL formulas is often achieved with a relatively simple temporal operator structure [173, 82].

Lemma 4.7. *The satisfiability problem of temporally safe HyperLTL is coRE-hard already for formulas of the form $\forall\exists^*.\square\psi$, where ψ only contains $\bigcirc$ operators.*

Proof. We reduce from the non-halting problem of deterministic Turing machines on the empty word, which is coRE-complete. We assume, w.l.o.g., that the tape of the Turing machine is left-bounded and only takes a step to the left when this is possible. We encode the position of the head with a proposition h. Throughout the construction, we maintain the invariant that h is set exactly once on all relevant traces. We cannot enforce this property directly as it would require to nest multiple $\square$ operators. We encode the alphabet Γ and the set of states Q with sets of atomic propositions S_Γ and S_Q. We enforce that in each step, exactly one proposition of each set is set to *true*.

$$\forall\pi.\square\Big(\bigvee_{a\in S_\Gamma}(a_\pi\land\bigwedge_{\substack{b\,\in\,S_\Gamma\\ b\neq a}}\neg b_\pi)\land\bigvee_{a\in S_Q}(a_\pi\land\bigwedge_{\substack{b\,\in\,S_Q\\ b\neq a}}\neg b_\pi)\Big)$$

We fix the current state to be the one that holds in the position of the head. Initially, the TM is in state q^0, the head at position 0, and the tape is blank. We use # for the blank symbol. We require that the initial configuration is present in the set.

$$\exists\pi.\,h_\pi\land q^0_\pi\land\square(\#_\pi\land\bigcirc\neg h_\pi)$$

We encode the possible transitions with a $\forall\pi.\,\exists\pi'$ formula. We ensure that if the configuration encoded by π is a valid one as described above, then the successor configuration is also valid. For correct transitions, all positions on π' that are not left or right of the head in π must remain unchanged. Second, the head must move either left or right and the symbol and state propositions are only allowed to change in the position of the old head.

$$\begin{aligned}\psi_{\text{LxorR}} \coloneqq \Big(&(\neg h_\pi\land\bigcirc\neg h_\pi\land\bigcirc\bigcirc\neg h_\pi)\to\bigcirc\bigwedge_{a\in AP}a_\pi\leftrightarrow a_{\pi'}\Big)\\ &\land\bigcirc h_\pi\to\Big(\bigcirc\neg h_{\pi'}\land(h_{\pi'}\oplus\bigcirc\bigcirc h_{\pi'})\land\big(\bigwedge_{a\in\Gamma}(a_\pi\leftrightarrow a_{\pi'})\land\bigcirc\bigcirc(a_\pi\leftrightarrow a_{\pi'})\big)\Big)\end{aligned}$$

In the formula above, $\oplus$ denotes the "exclusive or" operator. Now we translate each transition $(q,a),(q',a',L)\in\delta$ as follows. If the head moves right instead of left, we change the position of the $\bigcirc$ operator accordingly.

$$\psi_{\text{trans}_1} \coloneqq \bigcirc(h_\pi \wedge a_\pi \wedge q_\pi) \rightarrow h_{\pi'} \wedge q'_{\pi'} \wedge \bigcirc a'_{\pi'}$$

The final transition formula is the following.

$$\forall \pi. \exists \pi'. \square(\psi_{\text{LxorR}} \wedge (\psi_{\text{trans}_1} \vee \psi_{\text{trans}_2} \vee \ldots))$$

We only encode transitions in which $q' \notin F$, i.e., only those transitions that do not make the Turing machine halt. The conjunction of the above formulas can be easily transformed into a $\forall\exists^2$ formula with a single $\square$ and only $\bigcirc$ operators in the scope of $\square$. The encoded TM has an infinite non-halting run iff the conjunction of the above formulas has a satisfying model. The trace set might not only contain the witnessing run but also non-valid configurations or non-reachable configurations, which we can just ignore. □

From Lemma 4.6 and Lemma 4.7, we obtain the final theorem.

Theorem 4.8. *The satisfiability problem of temporally safe HyperLTL is coRE-complete.*

4.2 Temporal Liveness

The natural counterparts of safety properties are liveness properties, which postulate that "something good happens eventually". Similar to the case of hypersafety, hyperliveness as a fragment is not well-suited when studying satisfiability: any hyperliveness property is, by definition, satisfiable. Encouraged by the results on temporal safety, we study HyperLTL formulas whose body describes a liveness property.

Definition 4.2. A HyperLTL formula $Q\pi_1 \ldots Q\pi_n. \psi$ is a *temporal liveness* formula if ψ (interpreted as an LTL formula over $AP_{\pi_1} \cup \ldots \cup AP_{\pi_n}$) describes a liveness property.

As opposed to the safety case (cf. Lemma 4.3), temporal liveness and hyperliveness are incomparable, also when restricted to the $\forall^*\exists^*$ fragment. In temporal liveness, we can easily express falsity as $\forall\pi\forall\pi'. \diamondsuit(a_\pi \wedge \neg a_{\pi'})$, which is not hyperliveness. Conversely, the formula $\forall\pi\exists\pi'. \square(a_\pi \nleftrightarrow a_{\pi'})$ is hyperliveness (as we can always add more witness traces) but not expressible in temporal liveness (because of the $\square$ operator).

Analogous to Theorem 4.8, we consider the full fragment of temporal liveness. Unlike the fragment of temporal safety, the class of temporal liveness is Σ^1_1-hard.

Theorem 4.9. *The satisfiability problem is Σ^1_1-hard for temporal liveness HyperLTL formulas.*

To prove Theorem 4.9, we show that we can reduce every $\forall^*\exists^*$ HyperLTL formula to an equisatisfiable formula from the $\forall^*\exists^*$ temporal liveness fragment.

Theorem 4.10. *Let φ be a $\forall^*\exists^*$ HyperLTL formula. Then there is a computable $\forall^*\exists^*$ formula φ' such that φ' is a temporal liveness property, and φ and φ' are equisatisfiable.*

Proof. Let $\varphi = \forall \pi_1, \ldots, \pi_n. \exists \pi'_1, \ldots, \pi'_m. \psi$. We distinguish between two cases, based on the satisfiability of ψ. Whether ψ is satisfiable is decidable in PSPACE as ψ is an LTL formula. First, assume that ψ is unsatisfiable. Then $\psi \equiv$ *false* and φ is unsatisfiable as well (as T must be non-empty by the definition of HyperLTL →satisfiability). We choose φ' to be $\forall \pi, \pi'. \Diamond(a_\pi \wedge \neg a_{\pi'})$. Clearly, φ' is a temporal liveness formula and also unsatisfiable. In the second case, assume that ψ is satisfiable. The idea is to move the start position of the formula under a $\Diamond$ operator. We introduce a fresh atomic proposition $\dagger$ and ensure that all traces satisfy the liveness property $\Diamond(\dagger \wedge \bigcirc\Box\neg\dagger)$. The *unique* position where $\dagger \wedge \bigcirc\Box\neg\dagger$ holds (the last time that $\dagger$ is true) is then the "start position" to evaluate the formula. We define →page 22

$$\psi' := \Diamond\left(\bigwedge_{i=1}^{n+m} \dagger_{\pi_i} \wedge \left(\bigcirc\Box\bigwedge_{i=1}^{n+m} \neg \dagger_{\pi_i}\right) \wedge \psi\right)$$

and $\varphi' := \forall \pi_1, \ldots, \pi_n. \exists \pi'_1, \ldots, \pi'_m. \psi'$. As ψ is satisfiable, ψ' is a liveness property. Thus, φ' is a temporal liveness formula. We claim that φ is satisfiable iff φ' is satisfiable. For the first direction, assume that T is a model for φ. The model with $\dagger$ added to the first step of all traces satisfies φ'. For the other direction, let T be a model of φ'. We assume w.l.o.g. that there is no set $T' \subsetneq T$ such that T' is also a model for φ'. As T is a model of φ', there exists a skolem function $f : T^n \longrightarrow T^m$ that assigns the witness traces for every possible choice of the universal quantifiers. Let's represent T as a directed graph where every trace is a node, and if $f(s) = s'$ for any two tuples of traces s and s', and $t_i \in s$ and $t_j \in s'$ with $t_i, t_j \in T$, then we draw a directed edge from node t_i to node t_j. As T is minimal, the resulting graph is strongly connected. Otherwise, there would be a strongly connected component whose traces also satisfy φ'. Formula φ' enforces that for any $t_1, \ldots, t_{n+m}$ with $f(t_1, \ldots, t_n) = (t_{n+1}, \ldots, t_{n+m})$, $\dagger$ holds for the last time at a time point that is common for all traces $t_1, \ldots, t_{n+m}$. As the graph representing T is strongly connected, the last position where $\dagger$ holds is the same for all traces in T. Let i be this position. Then $\{t[i, \infty] \mid t \in T\}$ is a model of φ. □

Note that the above proof does not change the quantifier structure in the case that the body of the formula is satisfiable.

The definitions of liveness for trace properties (cf. →Definition 2.2) and hyperliveness (cf. →Definition 2.4) imply that a property is satisfiable. As demonstrated by Theorem 4.9, the same does not hold for temporal liveness hyperproperties. We can, however, identify a fragment within temporal liveness for which the intuition that liveness →page 19 →page 20

implies satisfiability transfers to the realm of hyperproperties. We say an LTL property ψ is a *deterministic liveness* property if it is a liveness property and can be recognized by a →deterministic Büchi automaton. →page 19

Proposition 4.11. *HyperLTL formulas of the form $\varphi = \forall\exists^*. \psi$, where ψ is a deterministic liveness property, are always satisfiable and have a finite model.*

Proof. Let $\varphi = \forall\pi_1. \exists\pi_2, \dots, \pi_n. \psi$ be such that ψ is quantifier-free and let $\mathcal{A}_\psi$ be a deterministic Büchi automaton for ψ over $AP_{\pi_1} \cup \dots \cup AP_{\pi_n}$. The crucial property we use is that for a deterministic liveness property, we can always reach an accepting state even after having read an arbitrary finite word. We claim that there always exists a finite model $T = \{t_1, \dots, t_n\}$. For any $i \in \{1, \dots, n\}$ define $f(i)$ as the tuple $(1, \dots, i-1, i+1, \dots, n)$. We construct $t_1, \dots, t_n$ such that if t_i is chosen for the universal trace, then $t_{f(i)[0]}, \dots, t_{f(i)[n-1]}$ can be picked for the existential traces. We iteratively construct a model as follows. Initially, we set $u_1 = \dots = u_n = \epsilon$. In the jth iteration, let $i = (j\%n) + 1$, i.e., we iterate through $1, \dots, n$. As $\mathcal{A}$ is a deterministic Büchi automaton describing a liveness property, there exist non-empty words $u'_1, \dots, u'_n$ of some common finite length such that $zip(u_i \cdot u'_i, u_{f(i)[0]} \cdot u'_{f(i)[0]}, \dots, u_{f(i)[n-1]} \cdot u'_{f(i)[n-1]})$ leads to an accepting state in $\mathcal{A}$. Continue the next iteration with each u_k extended with u'_k. We set $t_1, \dots t_n$ as the infinite traces constructed in the limit. By construction, for every $i \in \{1, \dots, n\}$, the unique run of the trace $zip(t_i, t_{f(i)[0]}, \dots, t_{f(i)[n-1]})$ visits an accepting state infinitely often. □

4.3 Finding Largest Models

To complement the decidability results from the previous sections, we propose a new (incomplete) algorithm to prove satisfiability and unsatisfiability of $\forall\exists^*$ HyperLTL formulas. So far, the only available algorithm checks for finite models of bounded size and then iteratively increases the bound [83]. The approach finds smallest models but cannot determine unsatisfiability. The insight for our algorithm is that $\forall\exists^*$ formulas are closed under union, therefore, a formula φ is satisfiable iff there is a (unique) *largest* model satisfying φ. To find the largest model of a formula, we iteratively eliminate choices for the $\exists^*$ quantifiers for which there are no witness traces when chosen as $\forall$ trace. Thereby, we do not only find largest models but can also detect unsatisfiability. Our incremental elimination is similar to a recent algorithm used in the context of finite-trace properties [33]. Both approaches have been developed independently.

4.3.1 Algorithm

Let $\varphi = \forall\pi. \exists\pi_1, \dots, \pi_n. \psi$ be a HyperLTL formula. Let $\mathcal{A}_\psi$ be a Büchi automaton over $AP_\pi \cup AP_{\pi_1} \cup \dots \cup AP_{\pi_n}$ which expresses ψ. We define $\mathcal{A}^\forall$ and $\mathcal{A}^{\exists i}$ as the automata

Algorithm 1 Algorithm that searches for the largest model of a $\forall\exists^n$ property. Initially, $\mathcal{A}$ is a Büchi automaton that accepts the body the HyperLTL property.

```
1: **procedure** FINDMODEL($\mathcal{A}$)
2:     **if** $\mathcal{L}(\mathcal{A}^{\forall}) = \emptyset$ **then**
3:         **return** UNSAT;
4:     **if** $\mathcal{L}(\mathcal{A}^{\exists i}) \subseteq \mathcal{L}(\mathcal{A}^{\forall})$ for all $1 \leq i \leq n$ **then**
5:         **return** SAT, model: $\mathcal{L}(\mathcal{A}^{\forall})$;
6:     $\mathcal{A}_{\text{new}} := \mathcal{A} \cap \mathcal{A}^{\forall}_{\pi_1} \cap \ldots \cap \mathcal{A}^{\forall}_{\pi_n}$;
7:     FINDMODEL($\mathcal{A}_{\text{new}}$);
```

(over AP) that existentially project $\mathcal{A}$ on the alphabet AP_π and AP_{π_i}, respectively. Our algorithm is depicted in Algorithm 1. Initially, we call FINDMODEL($\mathcal{A}_\psi$).

The first candidate is $\mathcal{A} = \mathcal{A}_\psi$. The automaton $\mathcal{A}^{\forall}_{\psi}$ accepts all words for which there exist witness traces for the existential quantifiers. If $\mathcal{L}(\mathcal{A}^{\forall}) = \emptyset$, then φ is unsatisfiable. If all potential witness traces in all $\mathcal{L}(\mathcal{A}^{\exists i})$ are contained in $\mathcal{L}(\mathcal{A}^{\forall})$ (so they have a witness trace themselves), φ is satisfiable and $\mathcal{L}(\mathcal{A}^{\forall})$ is a model. If neither is the case, we refine $\mathcal{A}$ by removing all traces that have an $\exists$-component that is not in $\mathcal{L}(\mathcal{A}^{\forall})$. We define $\mathcal{A}_{\text{new}}$ as the intersection $\mathcal{A} \cap \mathcal{A}^{\forall}_{\pi_1} \cap \ldots \cap \mathcal{A}^{\forall}_{\pi_n}$ where $\mathcal{A}^{\forall}_{\pi_i}$ is $\mathcal{A}^{\forall}$ with the alphabet changed from AP to AP_{π_i}. That means that we intersect each $\exists$-component of $\mathcal{A}$ with its $\forall$-component. We can compute $\mathcal{A}_{\text{new}}$ via a standard intersection construction on Büchi automata. Note that $\mathcal{A}_{\text{new}}$ might again produce witness traces that themselves do not have witness traces in the current automaton, so we continue recursively. The algorithm maintains the following invariant.

Proposition 4.12. *In every iteration of the algorithm it holds that $\mathcal{L}(\mathcal{A}_{new}) \subseteq \mathcal{L}(\mathcal{A})$, and for every trace set T with $T \models \forall\pi.\, \exists\pi_1, \ldots, \pi_n.\, \psi$, we have $T \subseteq \mathcal{L}(\mathcal{A}^{\forall})$.*

Using Proposition 4.12 it is easy to see the following.

Lemma 4.13. *Given a formula $\varphi = \forall\pi.\, \exists\pi_1, \ldots, \pi_n.\, \psi$, if Algorithm 1 terminates with UNSAT, the formula is unsatisfiable. If it terminates with SAT and model $\mathcal{L}(\mathcal{A}^{\forall})$, then $\mathcal{L}(\mathcal{A}^{\forall})$ is the unique largest model of φ.*

Models of $\forall^*\exists^*$ formulas are, in general, not closed under union, so our algorithm does not extend beyond $\forall\exists^*$.

4.3.2 Implementation and Experiments

We have implemented the algorithm in a tool called LMHyper (**L**argest **M**odels for **Hyper**LTL). LMHyper reads a $\forall\exists^*$ HyperLTL formula φ and searches for a proof of (un)satisfiability. Internally, we represent the current candidate model as a generalized

Table 4.1: Comparison of LMHyper and MGHyper on 100 random formulas. The time is reported as the average time spent on solved cases in ms, #Iterations is the average number of recursive calls needed by LMHyper. The timeout was 5sec.

	MGHyper		LMHyper		
Size of AST	**%Solved**	**Time**	**%Solved**	**Time**	**#Iterations**
15	95%	40	100%	235	0.38
16	93%	39	99%	239	0.44
17	95%	39	100%	221	0.43
18	92%	38	100%	201	0.39
19	95%	40	100%	180	0.43
20	97%	42	100%	215	0.27

Büchi automaton and use SPOT [71] to perform automata operations. The only other available tool for $\forall\exists^*$ HyperLTL satisfiability is MGHyper [83], which implements the incremental approach to find models of bounded size. We compare LMHyper against MGHyper on two benchmark sets.

Random Formulas. We created a set of random $\forall\exists^*$ formulas by sampling the LTL body of the formula using RANDLTL [71]. The results are given in Table 4.1. LMHyper usually takes longer than MGHyper but can handle a larger percentage of formulas. We observe that randomly generated HyperLTL formulas are, in most cases, satisfiable by a model with a single trace, as the atomic propositions are seldom shared between different trace variables. This explains the high success rate of MGHyper.

Infinite and Large Models. We compiled a small number of more interesting properties that do not have single-trace models. Our results for this set of this set of benchmarks are depicted in Table 4.2. The Infinite instance expresses that a model has infinitely many traces. The Enforce-n formulas enforce a models that have at least n traces. They are defined as

$$\forall \pi.\, \exists \pi_1, \ldots, \pi_n.\, \bigwedge_{i \neq j} \Diamond (a_{\pi_i} \not\leftrightarrow a_{\pi_j})$$

The Unsatisfiable-n specifications are unsatisfiable. They are defined as

$$\forall \pi.\, \exists \pi'.\, \neg a_\pi \,\mathcal{U} (a_\pi \wedge \bigcirc \Box \neg a_\pi) \wedge \bigcirc^n \Box \neg a_\pi \wedge \Diamond (a_\pi \wedge \bigcirc a_{\pi'})$$

The formula expresses that a holds exactly once on each trace and that no a occurs after n steps. However, it also requires that for every trace, there is another one where

Table 4.2: Comparison of LMHyper and MGHyper on hand-crafted formulas. We use ✓ if the specification is satisfiable and ✗ otherwise. The time is given in ms. #Iterations denotes the number of recursive calls needed by LMHyper. The timeout was 5min.

	MGHyper		LMHyper		
Instance	**Result**	**Time**	**Result**	**Time**	**#Iterations**
Infinite	-	TO	✓	350	1
Enforce-2	✓	444	✓	262	0
Enforce-3	-	TO	✓	334	0
Enforce-5	-	TO	✓	491	0
Unsatisfiable-3	-	TO	✗	777	3
Unsatisfiable-5	-	TO	✗	1363	5
Unsatisfiable-9	-	TO	✗	1681	9

a occurs one step later. The formula is designed such that Algorithm 1 requires *n* iterations to discover unsatisfiability. MGHyper times out for most of the examples; even on simple properties like Enforce-3. In contrast, LMHyper can verify properties enforcing many traces in a single iteration because the number of iterations is independent of the number of traces in a model. As expected, all Unsatisfiable-*n* formulas are discovered to be unsatisfiable, and LMHyper requires the expected number of iterations.

Chapter 5

Related Work

We relate the first part of the thesis to closely related work on temporal hyperlogics. The introduction (→Chapter 1) already discussed the foundational work which motivated this thesis; in this chapter, we want to go into more technical depth. We discuss the expressiveness of similar hyperlogics and their satisfiability problem. We also provide more background on the use of team semantics in various logical areas. →page 1

Plain Hyperlogics. In this part of the thesis, we focused on hyperlogics that express plain hyperproperties on trace sets containing ω-regular traces produced by standard finite-state transition systems. There are several such hyperlogics that we did not include in our hierarchy; for some, the relation to our logics is still unknown. One example is HyperPDL-Δ [121], which lifts Propositional Dynamic Logic [96] to hyperlogics. Its expressiveness falls between HyperQPTL and HyperLTL [121]. Another example is HyperCTL$^*_{lp}$ [34], which is a linear interpretation of the branching-time logic HyperCTL*. It was introduced to unify the expressiveness of LTL extended with the knowledge operator [126] and HyperLTL. It is not clear yet how this logic fits into the linear-time hierarchy, the author of this thesis expects it to fall strictly between HyperLTL and FO$[<, E]$ (see →Section 10.2.1 for more on future work). As an alternative to first-order logics with the equal-level predicate, Hypertrace Logic was proposed [21]. It is a two-sorted first-order logic, whose quantifiers distinguish between trace and time quantifiers. We strongly suspect that Hypertrace Logic and FO$[<, E]$ are expressively equivalent but are not aware of any published result in that direction. Beyond the realm of infinite traces, there exists HyperLDL$_f$ [114], which reasons about finite traces. →page 133

Hyperlogics with Additional Expressiveness. The interest in hyperlogics is particularly driven by privacy and information flow policies. It is therefore natural to design hyperlogics that can express asynchronous, quantitative, probabilistic, and timed hyperproperties. There is vast body of work on specification languages for hyperproperties, just because of the interest in information flow security alone. Here, we focus on temporal hyperlogics.

Asynchronous hyperproperties align the traces in a way that different positions on traces are related. This is necessary, for example, to specify stutter-equivalent properties or to relate systems with different clock cycles. H_μ [122] is a linear fixpoint hyperlogic that can advance time asynchronously on some of the quantified traces in each step. Asynchronous hyperlogics have also been obtained by labeling the relevant positions of a trace with an LTL formula [35, 27]. Another way to define asynchronous hyperlogics is to equip a logic with quantification over trajectories, which describe how to align the positions on the traces [22, 32, 136]. In a similar spirit, a very recent work obtains asynchronous hyperlogics based on team semantics by quantifying a so-called evaluation function [120]. A branching-time hyperlogic for asynchronous hyperproperties is HyperATL* [26], which extends the game-based temporal logic ATL*.

Quantitative hyperlogics are motivated by the need to quantify the amount of information that is leaked, for example to express declassification. Quantitative hyperlogics have been obtained based on model counting [89], there also exist non-temporal logics for the expression of quantitative hyperproperties [42, 204]. Probabilistic hyperproperties are hyperproperties that reason about the probabilities of certain events happening. Probabilistic temporal hyperlogics are therefore often evaluated on Markov chains [3, 228] or, to allow nondeterminism, on Markov decision processes [67, 2]. As a last class of hyperlogics, we want to mention logics for the expression of timed requirements, e.g., HyperMTL based on metric temporal logic [136, 32].

Satisfiability of Hyperlogics. Satisfiability is a particularly hard problem for hyperlogics. HyperLTL is subsumed by many logics studied in this thesis, so its high undecidability carries over to these logics. For HyperQPTL, propositional quantifiers are known to restrict the prefix of the trace quantifiers for which satisfiability is still decidable [123]. For example, while $\exists^*\forall^*$ HyperLTL formulas are decidable, they are not when preceded with universal propositional quantification, i.e., the $\forall_p^*\exists_\pi^*\forall_\pi^*$ fragment of HyperQPTL is undecidable. In HyperCTL*, the $\exists^*\forall^*$ fragment is undecidable as well, as the existential traces might occur in the scope of a $\square$ operator, which acts as a universal quantifier [123]. The general HyperCTL* satisfiability problem is Σ_1^2-complete [98]. For TeamLTL, it is known that TeamLTL$(\subseteq, \owedge)$ is Σ_1^0-hard, while TeamLTL$(\subseteq, \owedge, \mathsf{A})$ is Σ_1^1-hard [225]. TeamLTL together with Boolean negation $\sim$, the satisfiability problem is even equivalent to third-order arithmetic [168].

Team Semantics. The development of team semantics began with the introduction of Dependence Logic [220], which adds the concept of functional dependence to first-order logic by means of new atomic dependence formulas. During the past decade, team semantics has been generalized to propositional [232], modal [219], temporal

[152], and probabilistic [70] frameworks, and fascinating connections to fields such as database theory [128], statistics [61], real-valued computation [129], and quantum information theory [139] has been identified. In the modal team semantics setting, model checking and satisfiability problems have been shown to be decidable, see [134] for an overview of the complexity landscape. Expressiveness and definability of related logics is also well understood, see, e.g., [135, 150, 205]. The study of temporal logics with team semantics was initiated in [152], where team semantics for computational tree logic CTL was given. In [120], a new semantics for TeamLTL and TeamCTL* was proposed, which extends the logics to capture more asynchronous hyperproperties. TeamLTL($\sim$) has recently also been related to team-based first-order and second-order logics in a Kamp-like analysis [151].

Part II

Synthesizing Smart Contracts

Chapter 6

Preliminaries

In this part of the thesis, we study the synthesis of finite and infinite state systems from linear-time properties and hyperproperties. As this part is concerned with synthesis, we extend the preliminaries from the first part (cf. →Chapter 2) with definitions →page 18 for state machines that distinguish between inputs and outputs of a system. We also explicitly introduce infinite-state machines. Furthermore, we define temporal stream logic, a logic which was designed to enable efficient synthesis of systems with data from infinite domains. As a concrete challenge, we consider the synthesis of the temporal control flow of smart contracts. We therefore briefly introduce the Solidity features we use.

6.1 Reactive Synthesis

In task of reactive synthesis is to construct a reactive system from a given specification φ. A reactive system receives inputs from an environment and reacts with outputs such that the resulting traces satisfy the specification.

LTL and HyperLTL Realizability. The realizability problem formulates the synthesis problem in terms of a *strategy function*, which takes a finite sequence of inputs and reacts with an output. Let a set $AP = I \cup O$ be given. In this part, we always assume that $I \cap O = \emptyset$. A strategy over AP is a function $\sigma\colon (2^I)^+ \to 2^O$ that maps sequences of input valuations 2^I to an output valuation 2^O. Strategies can thus be arranged as *strategy trees* as illustrated in Figure 6.1. Given an input sequence $w = w_0 w_1 w_2 \cdots \in (2^I)^\omega$, the trace corresponding to a strategy σ is defined as $w \cup \sigma(w) := (w_0 \cup \sigma(w_0))(w_1 \cup \sigma(w_0 w_1)) \ldots \in (2^{I \cup O})^\omega$. Given a strategy σ, we denote by *traces*(σ) the set $\{w \cup \sigma(w) \mid w \in (2^I)^\omega\}$. We say that a strategy σ satisfies an LTL formula ψ if for every $t \in$ *traces*(σ), it holds that $t \models \psi$. It satisfies a HyperLTL formula φ if *traces*$(\sigma) \models \varphi$. A strategy function is not necessarily finitely

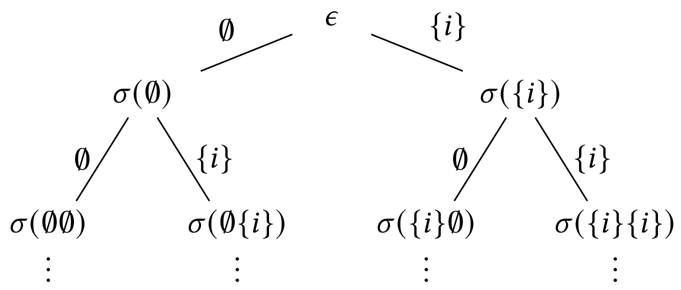


Figure 6.1: The strategy tree for an LTL strategy σ branching on a single input i.

representable. Compared to the realizability problem, the *synthesis problem* asks for a finite representation of a strategy, e.g., in form of a Mealy machine.

Mealy Machines. We represent reactive systems with Mealy machines [179], which separate the alphabet into input and output symbols, i.e., $\Sigma = \Sigma_{in} \cup \Sigma_{out}$ and $\Sigma_{in} \cap \Sigma_{out} = \emptyset$. A Mealy machine is a tuple $(S, s_0, \Sigma_{in}, \Sigma_{out}, \delta)$, where S is a set of states, s_0 is the initial state, Σ_{in} is the input alphabet, Σ_{out} is the output alphabet, and $\delta \subseteq S \times \Sigma_{in} \times \Sigma_{out} \times S$ is the transition relation. We mostly call Mealy machines just state machines. We call a state machine $\mathcal{M}$ *finite-state machine* if both Σ and S are finite, and *infinite-state machine* otherwise. An infinite sequence $t \in \Sigma^\omega$ is a *trace* of $\mathcal{M}$ if there is an infinite sequence of states $r \in S^\omega$ such that $r[0] = s_0$ and $(r[i], t[i]_{|\Sigma_{in}}, t[i]_{|\Sigma_{out}}, r[i+1]) \in \delta$ for all points in time $i \in \mathbb{N}$. A finite sequence of states $r \in S^+$ results in a finite trace $t \in \Sigma^+$. We denote the set of all traces of $\mathcal{M}$ by *traces*$(\mathcal{M})$ and the set of all finite traces by *finTraces*$(\mathcal{M})$. We define Mealy machines to be possibly nondeterministic; the transition relation is therefore a relation, not a function. A Mealy machine $\mathcal{M}$ *implements a strategy* if, in each state, there is exactly one outgoing transition for every valuation of input variables, i.e., if δ is a function of type $S \times \Sigma_{in} \rightarrow \Sigma_{out} \times S$.

Safety Games. In the second part of this work, we frequently consider properties that are →safety properties representable by →safety automata. A safety automaton over $\Sigma = \Sigma_{in} \cup \Sigma_{out}$ induces a safey game, where the environment player and the system player take turns in choosing the next input symbol and output symbol, respectively. Each pair of input and output symbol together determines the next transition. The *winning region* $S_{\mathcal{W}} \subseteq S$ is the maximal subset of states such that in every state, there exists at least one transition into said subset for every input symbol. The winning region can be computed by determining the fixpoint of the following set of equations.

→page 19
→page 19

$$Attr^0(S) = S$$
$$Attr^{i+1}(S) = Attr^i(S) \setminus noSucc(Attr^i(S))$$
$$\text{where } noSucc(S') := \{s \in S' \mid \exists A_{in} \in \Sigma_{in}.\ \neg\exists A_{out} \in \Sigma_{out}.\ (s, A_{in}, A_{out}, s') \in \delta \land s' \in S'\}$$

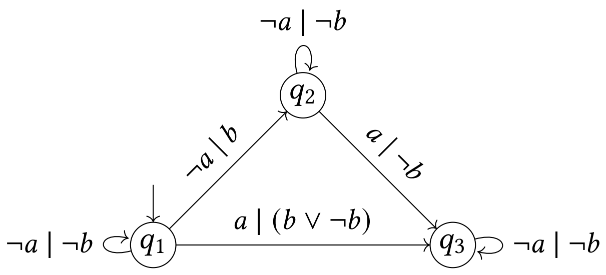


Figure 6.2: Winning region for assumption $\square(a \rightarrow \bigcirc\square\neg a)$ and guarantee $\square((b \rightarrow \bigcirc\square\neg b) \wedge (a \rightarrow \bigcirc\square\neg b))$.

The winning region of a safety game can be computed in linear time in the size of the automaton [64]. The property represented by the safety automaton is realizable iff the winning region contains the initial state. Note that a winning region does not necessarily implement a strategy, as a state might contain several transitions for the same input symbol. To obtain a strategy from the winning region, one can simply fix a transition for every input.

Many of our specifications are contain assumptions, which restrict the allowed input sequences, and guarantees. When depicting winning regions of such specifications, we only depict the transitions of the winning region whose inputs are allowed by the assumption. That means we do not include the “winning sink state” that the systems enters once an assumption is violated. As an example, consider the specification with input a, output b, assumption $\square(a \rightarrow \bigcirc\square\neg a)$, and guarantee $\square((b \rightarrow \bigcirc\square\neg b) \wedge (a \rightarrow \bigcirc\square\neg b)$. The winning region for this example, which is depicted in Figure 6.2, contains all traces that satisfy both assumption and guarantee. We use the symbol | to separate inputs and outputs in the transition labels.

6.2 Temporal Stream Logic

We introduce the syntax and semantics of TSL [93] following [91, 93]. TSL extends LTL with the concept of cells, which hold data from an arbitrary domain. To abstract from concrete data points, TSL employs uninterpreted functions and predicates. A TSL formula describes a system that receives a stream of inputs, abstracts from the concrete values using predicates, and produces a stream of cell updates using function applications. We formally define values, functions, and predicates. The set of all values is denoted by $\mathcal{V}$, the Boolean values by $\mathbb{B} \subseteq \mathcal{V}$. An n-ary function $f : \mathcal{V}^n \rightarrow \mathcal{V}$ computes a value from n values. An n-ary predicate $p : \mathcal{V}^n \rightarrow \mathbb{B}$ assigns a Boolean value to n values. The sets of all functions and predicates are denoted by $\mathcal{F}$ and $\mathcal{P} \subseteq \mathcal{F}$, respectively. Constants are both 0-ary functions and values.

Let $\mathbb{I}$ and $\mathbb{C}$ be the set of inputs and cells, and let Σ_F and $\Sigma_P \subseteq \Sigma_F$ be the set of function and predicate symbols. *Function terms* τ^f are recursively defined by

$$\tau^f ::= \mathsf{s} \mid f\ \tau^f \ldots \tau^f$$

where $\mathsf{s} \in \mathbb{I} \cup \mathbb{C}$ and f is an n-ary function symbol. *Predicate terms* τ^p are obtained by applying a predicate to function terms. The sets of all function and predicate terms are denoted by $\mathcal{T}_F$ and $\mathcal{T}_P \subseteq \mathcal{T}_F$, respectively. TSL formulas are built according to the following grammar:

$$\psi ::= \neg\psi \mid \psi \wedge \psi \mid \bigcirc\psi \mid \psi\,\mathcal{U}\,\psi \mid \tau^p \mid [\![\mathsf{c} \leftarrowtail \tau^f]\!]$$

where $\mathsf{c} \in \mathbb{C}$, $\tau^p \in \mathcal{T}_P$, and $\tau^f \in \mathcal{T}_F$. An *update term* $[\![\mathsf{c} \leftarrowtail \tau_f]\!]$ denotes that the value of function term τ_f is assigned to cell c. We denote the set of update terms with $\mathcal{T}_U$. As for LTL, we also use the derived operators $\Box$, $\Diamond$, and $\mathcal{W}$.

Function and predicate terms are syntactic objects, i.e. the term $p(f\ \mathsf{c})$ only becomes meaningful when we assign, for example, c to the value 3, f to the function $f(x) = x + 1$ and p to $p(x) = x > 0$. To assign a semantic interpretation to function and predicate symbols, we use an *interpretation* $\langle\cdot\rangle : \Sigma_F \to \mathcal{F}$. A cell assignment $\mathbb{C} \to \mathcal{T}_F$ is a total function assigning exactly one function term to each cell. The set of all assignments $\mathbb{C} \to \mathcal{T}_F$ is denoted by C. A *computation* $\varsigma \in C^\omega$ describes the control flow of the cells, i.e., which function term is associated with a cell at which point in time. For every cell $\mathsf{c} \in \mathbb{C}$, let $init_\mathsf{c}$ be a designated value assigned initially to c. Input streams $\mathcal{I}^\omega$ are infinite sequences of assignments of inputs to values.

Given a computation ς, an input stream ι, and a point in time i, we can evaluate a function term τ^f. The *evaluation function* $\eta_{\langle\cdot\rangle} : C^\omega \times \mathcal{I}^\omega \times \mathbb{N} \times \mathcal{T}_F \to \mathcal{V}$ is defined as

$$\eta_{\langle\cdot\rangle}(\varsigma, \iota, i, \mathsf{s}) := \begin{cases} \iota\ i\ \mathsf{s} & \mathsf{s} \in \mathbb{I} \\ init_\mathsf{s} & \mathsf{s} \in \mathbb{C} \wedge i = 0 \\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, i{-}1, \varsigma\ (i{-}1)\ \mathsf{s}) & \mathsf{s} \in \mathbb{C} \wedge i > 0 \end{cases}$$
$$\eta_{\langle\cdot\rangle}(\varsigma, \iota, i, f\ \tau_0 \ldots \tau_n) := \langle f\rangle\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, i, \tau_0)\ \ldots\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, i, \tau_n)$$

Note that $\iota\ i\ \mathsf{s}$ denotes the value that input stream ι assigns to input s at position i. Likewise, $\varsigma\ i\ \mathsf{s}$ is the function term that ς assigns to cell s at point in time i. As an example, to compute the value of cell x in step i, we might obtain from the computation that x is updated to $f\ \mathsf{x}$ in step i, so we recursively evaluate x in step $i - 1$ and apply the function assigned to f to the result. We evaluate a TSL formula ψ with respect to an assignment function $\langle\cdot\rangle$, an input stream $\iota \in \mathcal{I}^\omega$, a computation $\varsigma \in C^\omega$, and a time step $i \in \mathbb{N}$. In most cases, the semantics is very similar to LTL. The only difference is that predicate terms are evaluated with the evaluation function, and update terms are

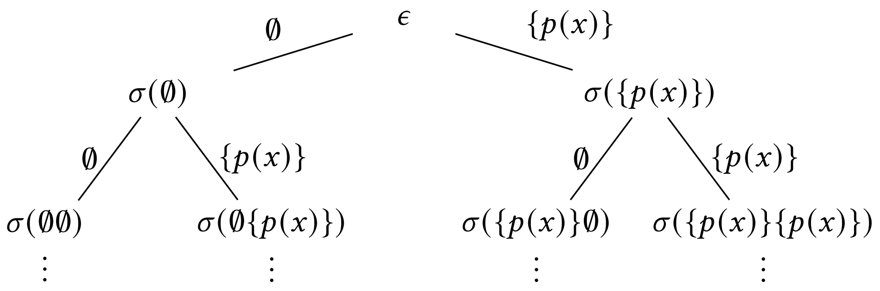


Figure 6.3: The strategy tree for a TSL strategy σ branching on a single predicate term $\mathcal{T}_P = \{p(x)\}$.

true if they occur syntactically in the computation.

$$
\begin{array}{lll}
\varsigma, \iota, i \models_{\langle\cdot\rangle} \neg\psi & \text{iff} & \varsigma, \iota, i \not\models_{\langle\cdot\rangle} \psi \\
\varsigma, \iota, i \models_{\langle\cdot\rangle} \psi_1 \wedge \psi_2 & \text{iff} & \varsigma, \iota, i \models_{\langle\cdot\rangle} \psi_1 \text{ and } \varsigma, \iota, i \models_{\langle\cdot\rangle} \psi_2 \\
\varsigma, \iota, i \models_{\langle\cdot\rangle} \bigcirc\psi & \text{iff} & \varsigma, \iota, i+1 \models_{\langle\cdot\rangle} \psi \\
\varsigma, \iota, i \models_{\langle\cdot\rangle} \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \exists j \geq i.\, \varsigma, \iota, j \models_{\langle\cdot\rangle} \psi_2 \text{ and } \forall i \leq k < j.\, \varsigma, \iota, k \models_{\langle\cdot\rangle} \psi_1 \\
\varsigma, \iota, i \models_{\langle\cdot\rangle} [\![\mathsf{c} \leftarrowtail \tau]\!] & \text{iff} & \varsigma\ i\ \mathsf{c} \equiv \tau \\
\varsigma, \iota, i \models_{\langle\cdot\rangle} p\ \tau_1 \ldots \tau_n & \text{iff} & \eta_{\langle\cdot\rangle}(\varsigma, \iota, i, p\ \tau_1 \ldots \tau_n)
\end{array}
$$

We use $\equiv$ to syntactically compare two terms. An *execution* (ς, ι) satisfies a TSL formula ψ if $\varsigma, \iota, 0 \models_{\langle\cdot\rangle} \psi$ holds (written $\varsigma, \iota \models_{\langle\cdot\rangle} \psi$). A TSL formula ψ is *satisfiable* iff there exists an interpretation $\langle\cdot\rangle$ and an execution (ς, ι) such that $\varsigma, \iota \models_{\langle\cdot\rangle} \psi$ holds [90].

TSL realizability. The realizability problem of a TSL formula ψ asks whether there exists a strategy which reacts to predicate evaluations with cell updates according to ψ. Formally, a TSL strategy is a function $\sigma : (2^{\mathcal{T}_P})^+ \to C$. The TSL strategy tree, depicted in Figure 6.3 for the example of a single predicate, looks very similar to an LTL strategy tree. The difference is that the branches are labeled with a set of predicate terms that are set to true by the environment in the respective step.

Given a strategy σ and an input stream ι, we can define the resulting computation $\sigma(\iota)$. To compute the cell updates at point in time i, we require the current inputs as well as the history of cell updates:

$$\sigma(\iota)(i) \coloneqq \sigma(\, \{\tau^p \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(\sigma(\iota), \iota, 0, \tau^p)\} \ldots \{\tau^p \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(\sigma(\iota), \iota, i, \tau^p)\}\,)$$

Note that in order to define $\sigma(\iota)(i)$, the definition uses $\sigma(\iota)$. This is well-defined, since the evaluation function $\eta_{\langle\cdot\rangle}(\varsigma, \iota, i, \tau)$ only uses $\varsigma\ 0 \ldots \varsigma\ (i-1)$.

Definition 6.1 (TSL realizability [93])**.** A TSL formula ψ is *realizable* iff there exists a strategy $\sigma : (2^{\mathcal{T}_P})^+ \to C$ such that for every input stream $\iota \in \mathcal{I}^\omega$ and every assignment function $\langle\cdot\rangle : \Sigma_P \to \mathcal{F}$, it holds that $\sigma(\iota), \iota \models_{\langle\cdot\rangle} \psi$.

Past-time TSL. TSL can be extended with past-time temporal operators. As for LTL, these do not extend the expressiveness of the logic but allow for a more concise formulation of properties. A definition of LTL with past-time operators can be found, e.g., in [48, 132]. We use operators *yesterday* $\ominus$, *weak yesterday* $\circleddash$, *historically* $\boxminus$, *once* $\diamondminus$, *since* $\mathcal{S}$, which are defined as follows.

$$\varsigma, \iota, i \models_{\langle\cdot\rangle} \ominus\psi \quad \text{iff} \quad \begin{cases} \varsigma, \iota, i-1 \models_{\langle\cdot\rangle} \psi & \text{for } i > 0 \\ \mathit{false} & \text{else} \end{cases}$$

$$\varsigma, \iota, i \models_{\langle\cdot\rangle} \psi_1 \mathcal{S} \psi_2 \quad \text{iff} \quad \exists j \leq i.\, \varsigma, \iota, j \models_{\langle\cdot\rangle} \psi_2 \wedge \forall j < k \leq i.\, \varsigma, \iota, k \models_{\langle\cdot\rangle} \psi_1$$

Similar to future-time operators, $\boxminus$ and $\diamondminus$ can be defined in terms of the since operator as $\diamondminus\psi = \mathit{true}\,\mathcal{S}\,\psi$ and $\boxminus\psi = \neg\diamondminus\neg\psi$. The $\circleddash$ operator is defined as $\circleddash\psi = \neg\ominus\neg\psi$. Unlike the "strong" yesterday operator, $\circleddash$ evaluates to *true* in the first step.

We denote by pastTSL (and pastLTL, respectively) the logic that only contains past-time operators except for an optional $\square$ operator at the highest level. Formulas without a leading $\square$ operator are called *initial formulas*; formulas with a $\square$ operator *invariants*. A computation ς and an input stream ι satisfy an initial pastTSL formula ψ if $\varsigma, \iota, 0 \models_{\langle\cdot\rangle} \psi$ holds. They satisfy an invariant ψ if $\varsigma, \iota, i \models_{\langle\cdot\rangle} \psi$ holds for all points in time $i \in \mathbb{N}$. PastTSL formulas can also be evaluated on finite traces as its formulas only depend on the past of a trace. For finite input streams and computations, ι and ς satisfy an invariant ψ if $|\iota| = |\varsigma| = n$ and $\varsigma, \iota, i \models_{\langle\cdot\rangle} \psi$ holds for all $i \in \mathbb{N}$ with $0 \leq i < n$.

Approximation of TSL in LTL. We quickly recap the approximation of TSL in LTL from [93]. The main idea is to forget the fact that predicates must evaluate to the same value if evaluated on cells that hold the same value. Then, every predicate τ^p and update term $[\![\mathsf{x} \leftarrowtail \tau^f]\!]$ can be replaced with an atomic proposition a_{τ^p} and $a_{\mathsf{x\_to\_}\tau^f}$, respectively, which is denoted by *syntacticConversion*(ψ) for a given TSL formula ψ. For example, $\square(p(\mathsf{x}) \to [\![\mathsf{x} \leftarrowtail f(\mathsf{x})]\!])$ translates to $\square(a_{p\_\mathsf{x}} \to a_{\mathsf{x\_to\_f\_x}})$. We call the set of atomic propositions obtained in that translation AP_ψ. Atomic propositions for predicate terms are inputs, propositions for update terms are outputs. Additionally, one ensures that in each step, exactly one update proposition holds for each cell.

$$\mathit{cellProps}(\varphi) := \square\Big(\bigwedge_{\mathsf{c}\in\mathbb{C}} \bigvee_{\tau\in\mathcal{T}_U^{\mathsf{c}}} \big(a_\tau \wedge \bigwedge_{\tau'\in\mathcal{T}_U^{\mathsf{c}}\setminus\{\tau\}} \neg a_{\tau'}\big)\Big)$$

where $\mathcal{T}_U^{\mathsf{c}}$ is the set of all update terms of cell c and a_τ is the atomic proposition associated with term τ.

We define the approximating LTL formula $[\psi]_{\text{atomic}}$ as follows.

$$[\psi]_{\text{atomic}} \coloneqq \mathit{syntacticConversion}(\psi) \wedge \mathit{cellProps}(\psi)$$

What gets lost in the approximation is the fact that terms that evaluate to the same value according to the TSL semantics must also have the same value in the LTL approximation. For example, after the update $[\![\mathtt{x} \leftarrowtail \mathtt{y}]\!]$ is performed, the predicate terms $p(\mathtt{x})$ and $p(\mathtt{y})$ must be assigned the same value in the next step. This behavior cannot be fully captured in LTL; the two atomic propositions $a_{p\_\mathtt{x}}$ and $a_{p\_\mathtt{y}}$ can be assigned different values by a strategy. The approximation is thus sound but not complete: any strategy for the LTL formula is also one for the TSL formula, but counterstrategies of the environment may be spurious.

6.3 Solidity

Most of the techniques we present in this part of the thesis are not limited to smart contracts. We therefore use only few specific Solidity features, mostly in the translation of our state machines to Solidity. We briefly introduce these features and refer to the Ethereum [101] and Solidity [102] documentations for details.

Solidity is a statically typed, mostly object-oriented programming language that reads like C++ but implements specific constructs for the development of smart contracts. Solidity uses methods and fields to store data. Methods can be labeled as `public` or `private`, where `public` is the default. Accounts on the Ethereum blockchain (which are either users or contracts) can call all public methods of a contract. If a method is labeled `payable`, the caller can attach *Ether* (the native cryptocurrency of Ethereum) to the call, e.g., to pay for an item in an auction. The current caller of the method can be accessed by `msg.sender`, the attached value by `msg.value`. Especially important is the rollback functionality `revert()`, which undos all effects of the current call. It is used, e.g., to implement preconditions of a method.

Every execution that changes the state of a smart contract costs *gas*. When calling a smart contract, the caller has to provide the amount of gas of needed for the transaction. Every unit of gas has a fixed price in Ether. When implementing a smart contract, it is therefore important to keep the cost of gas low. Operations with high gas costs, are, for example, lookups in a mapping.

Chapter 7

Parameterized Synthesis of Smart Contracts

Smart contracts are small programs that typically implement auctions, voting protocols, digital coins, marketplaces, or asset transfers. As such, smart contracts often involve monetary transactions between the parties. Recent history has witnessed numerous bugs in smart contracts, some of which led to substantial monetary losses. One critical area is the implicit state machine of a contract: to justify the removal of a trusted third party – a major selling point for smart contracts – all parties must trust that the contract indeed enforces the agreed order of transactions. Formal methods provide formal guarantees and thus play a major role in these efforts, also because of the relatively small size of most smart contracts. Indeed, the *code is law* paradigm is more and more shifting towards a *specification is law* paradigm [9]. Formal verification has been applied successfully to prove the correctness of the underlying state machine of existing smart contracts [188, 215, 227, 184].

Synthesis, i.e., the automatic construction of Solidity code *directly* from a temporal specification, has received only little attention so far. The idea to employ synthesis as an alternative to verification applies especially in the context of smart contracts. Many contracts belong to some standardized class like the ERC20 token standard. Whenever the contract is extended with new functionality, the contract has to be verified again. Synthesis, in contrast, is a property-oriented approach and enables an incremental and modular development process: specifications can be easily extended with additional conjuncts for new functionality. The implications of the extension on the state machine are handled by the synthesis algorithm.

Recent work [91] proposed to synthesize the control flow graph of a smart contract from the past-time fragment of temporal stream logic. TSL extends linear-time temporal logic (LTL) with the concept of fields together with uninterpreted functions and predicates. The uninterpreted predicates facilitate reasoning about the control flow of the contract and its fields as they store data from potentially infinite domains. For example,

$$\square(\texttt{close} \rightarrow \texttt{sender} = \texttt{owner()} \land \texttt{numberVotes} > \texttt{cNum()})$$

expresses the invariant that method `close` can only be called by the owner of the contract and only if the value of field `numberVotes` is larger than some constant threshold `cNum()`. As demonstrated in [91], temporal control flows of smart contracts can be specified as safety properties expressed in pastTSL. This leads to efficient symbolic algorithms, which are implemented in the tool SCSynt. The tool also translates the synthesized state machine into Solidity code.

In this chapter, we extend pastTSL synthesis to improve its usefulness for smart contracts. First, building on the efficiency of the approach, we embed the synthesis in a feedback loop that finds potential specification errors. In particular, it warns the developer about free choices (which often indicate that the system is under-specified) and potential deadlocks. Second, we observe that many smart contract specifications need to distinguish between calls to the same method but with differently assigned parameters. ERC20 token systems, for example, do not have a single global control flow but instead a "local" control flow for each address. If address `n` wants to spend `k` tokens of `m`'s account, `m` must have approved that `n` may spend at least that many tokens. It is not possible to express such properties directly in TSL. We therefore extend TSL with the concept of universally quantified parameters. The following formula expresses the above property, where `transferFrom` is a method parameterized with `m` and `n`, and `approved` is a field mapping from two addresses to a positive integer.

$$\forall \texttt{m}.\, \forall \texttt{n}.\, \square(\texttt{transferFrom(m,n)} \rightarrow \texttt{approved(m,n)} \geq \texttt{arg@amount})$$

Above, `arg@amount` denotes an additional parameter of `transferFrom` that does not have to be quantified as is never occurs as a field parameter. Due to the universal quantification, the formula describes an infinite-state machine. This poses the challenge to synthesize a finite representation of the system that can be translated into a Solidity contract with moderate gas consumption. To do so, we represent the infinite-state machine with a hierarchical structure of small finite-state machines that communicate with each other.

We developed this work with Solidity in mind as a step towards a less error-prone development of smart contracts. Our approach generalizes to a broader class of programs, however. The idea of specifying the temporal control flow of a program with parameterized pastTSL is not specific to smart contracts or Solidity. Neither is the general workflow that embeds the synthesis in an analysis of the state machine with respect to free choices and deadlocks.

Outline. We first recap the synthesis of smart contract control flow graphs from pure pastTSL [91]. In Section 7.2, we present the workflow that embeds the synthesis into an analysis of the specification. Section 7.3 defines parameterized pastTSL and illustrates the use of the logic on the examples of a voting protocol and an ERC20 token contract. Subsequently, in Section 7.4, we discuss how to synthesize Solidity smart contracts from parameterized specifications, building on the synthesis from pure pastTSL. Lastly, Section 7.5 discusses our implementation of the presented approaches as extension of SCSynt. To evaluate our approach, we specify and synthesize various smart contracts. We compare the gas consumption of our synthesized contracts with other approaches that produce formally verified contracts and also with manually written contracts.

Publications. This chapter builds on the following paper, which is at the point of submitting this thesis under peer-review. The paper introduces the idea to use pastTSL for smart contract synthesis as well as the extensions described in this chapter. Only the latter is a contribution of this thesis.

[91] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. **Reactive Synthesis of Smart Contract Control Flows**. *arXiv 2022*. URL: `https://arxiv.org/abs/2205.06039`.

7.1 Recapitulation: Synthezing Smart Contract State Machines from PastTSL

In this section, we recap the approach presented in [91], which proposes to specify the temporal behavior of smart contracts in the past-time fragment of TSL. This enables an efficient BDD-based synthesis, which is implemented in SCSynt. The idea of the approach is to focus on the specification of the underlying state machine of a smart contract. The synthesized state machine is translated to Solidity code, which forms the backbone of the contract. The code can afterwards be augmented with additional functionality if needed. As long as no assumption formulated in the specification is violated in the process, the resulting contract will have a correct temporal control flow.

7.1.1 Specification

The developer specifies the contract in pastTSL, a fragment of TSL which describes invariants over the history of a trace up to the current point. TSL is a convenient logic to not only describe the correct order of method calls but also how fields need to be updated to ensure a correct control flow. This can be achieved through TSL's cells and uninterpreted functions and predicates. To avoid confusion, we call Solidity functions

```
1  Methods: vote, close, reveal
2  Fields: voters
3  Functions: ∈, add
4  Predicates: >, =
5  Constants: owner(), cTime()
6  Inputs: time, sender
7
8  --- Assumptions ---
9  □(⊖ time > cTime() → time > cTime());
10
11 --- Requirements ---
12 □(close → sender = owner());
13 □(vote → ¬(sender ∈ voters));
14 □(vote → ¬(time > cTime()));
15 □(close → time > cTime());
16 □(close → ∼⃝ ⊟ ¬close);
17 □(reveal → ⟐ close);
18
19 --- Obligations ---
20 □(vote → ⟦voters ↢ add(sender, voters)⟧);
21 □(¬vote → ⟦voters ↢ voters⟧);
```

Specification 7.1: Simple voting specification in pastLTL.

methods and reserve the notion of *functions* for TSL. We use TSL's cells to describe the data flow of contract's fields and use uninterpreted predicates to state access rights and other requirements on the values of the methods' parameters. In the context of smart contexts, we therefore use "cell" and "field" interchangeably. The TSL specification abstracts from the concrete implementation of functions and predicates and instead states when a method may be called and how fields have to be updated depending on the valuation of the predicates.

We illustrate the idea with a simple specification of a voting contract given in Specification 7.1. The voting contract will serve (in various forms) as running example in this and the following chapter. Method calls are modeled with Boolean inputs (e.g., `vote`). Formally, Boolean inputs do not exist in TSL, `vote` is thus syntactic sugar for a predicate over an input, e.g., `isVote(methodinput)`. Checks on a method's arguments (e.g., `sender = owner()`) use uninterpreted predicates (= in this case). The assignments of the contract's fields are described as updates. A specification is divided into three parts: assumptions, requirements, and obligations.

Assumptions provide information on the uninterpreted predicates in the specification. In Specification 7.1, we use the assumption that time is monotone, i.e., if it passes a threshold like the constant `cTime()`, it stays greater than the threshold (l. 9). Assumptions also constitute a way to interact with deductive verification tools for smart contracts. If the developer is confident that they can prove some pre- and postconditions of a method, this can be added as an assumption. As an example, we could extend the voting specification with the following formulation of a Hoare triple.

$$\square(\texttt{numberOf}(\texttt{voters}) > 10 \land \texttt{vote} \rightarrow \bigcirc(\texttt{numberOf}(\texttt{voters}) > 10))$$

The above formula states if there has been more than 10 voters, then an additional vote will not decrease that number. Assumptions added to the formula need to be enforced outside the synthesis algorithm. Hoare triples like the one above can be proven using deductive verification tools like Celestial for smart contracts [66]. Other assumptions are made true by the implementation of a predicate, e.g., a transitivity assumption on the $>$ predicate. In addition to the assumptions provided by the developer, SCSynt adds the assumption that only one method is called at a time. This assumption is enforced in the translation to Solidity using a Boolean flag, which also prevents reentrancy attacks [174].

Requirements define the allowed sequence of method calls based on the history of method calls and on predicates on the fields and method arguments. In Specification 7.1, we state, for example, that `reveal` can only happen after `close` has been called (l. 17). Lastly, *obligations* describe when and how fields need to be updated. In the above specification, we require that the current sender is added to the list of voters when `vote` is called (l. 20). Otherwise, `voters` has to remain unchanged (l. 21).

7.1.2 Synthesis and Translation to Solidity

Given the specification, the tool SCSynt synthesizes a state machine implementing a strategy that satisfies the specification (if there is any). While TSL synthesis is in general undecidable [93], correct solutions can be found through approximation in LTL, which works very well in practice [93, 112, 92]. SCSynt follows the approximation described in [93], extended with past-time temporal operators. The resulting formula is a past-time LTL formula that describes a →safety property. For past-time LTL formulas, synthesis is very efficient for two reasons. First, to evaluate a pastLTL formula at point in time i, one only needs to know the evaluation of all subformulas at point $i-1$ [132]. This makes it possible to construct a pastLTL formula's →safety automaton with a linear traversal of the formula. The resulting →safety game can also be solved in linear time with respect to the automaton [64]. SCSynt implements the computation of the winning region symbolically using BDDs to represent the safety automaton. It also minimizes the resulting winning region. If the specification is realizable, the tool returns a strategy. The only realizing strategy for the voting specification is depicted as state machine in Figure 7.1.

→page 19
→page 19
→page 76

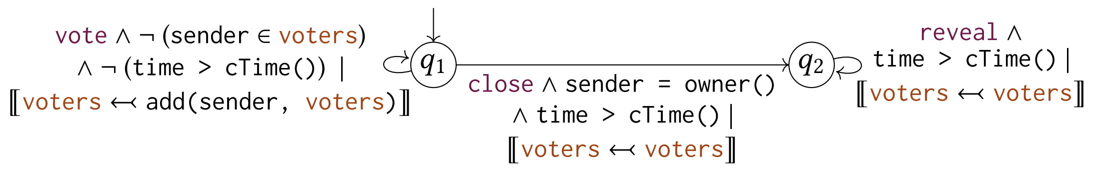


Figure 7.1: Synthesized state machine for Specification 7.1.

```
function vote(uint _choice) public {
  if(currState == <source state of t1> && <guard of t1>){
    currState = <target state of t1>
    <field updates of t1>
  } else if(currState == <source state of t2> && <guard of t2>)
     {
    currState = <target state of t2>
    <field updates of t2>
  } else if ...
  else{
    revert();
  }
}
```

Solidity Code 7.1: Sketch of generated Solidity code for method vote.

The state machine implementing the strategy is automatically translated to Solidity code. Solidity Code 7.1 shows the skeleton of the code generated for the vote method. The developer only needs to provide a signature which indicates, e.g., the types of fields and methods. Furthermore, they must implement the (usually simple) uninterpreted functions and predicates. Common functions and predicates such as addition + and the > predicate are translated automatically. Further keywords are value, sender, balance, and owner, which are translated to msg.sender, msg.value, address(this).balance, and an owner field which is set in the constructor. The if statements implement the transition system logic. If none of the guards evaluates to true, the call to the method is reverted. The signature of the method has a _choice parameter, which stands for the actual vote of the caller. In our example specification, we did not specify how the handle the vote itself and how to ensure that the correct winner is chosen, this must still be implemented by hand.

7.2 Free Choices and Deadlock Detection

We extend the synthesis described in the last section with an analysis for potential specification errors. This kind of feedback loop is made possible by the efficiency of

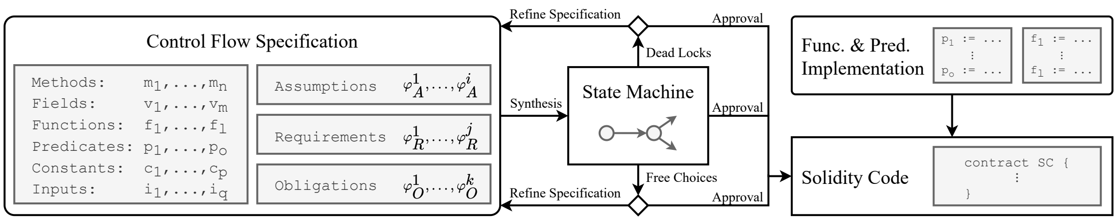


Figure 7.2: Synthesis workflow with analysis regarding free choices and potential deadlocks.

the synthesis that allows the developer to go back and fix the specification and obtain timely feedback. Our analysis searches for free choices and potential deadlocks in the synthesized state machine. The resulting workflow from the developer's perspective is depicted in Figure 7.2.

Free choices indicate that the specification is realizable by multiple strategies. In many cases, this is not intended and means that the contract is under-specified. A specification admits multiple strategies if in some state of the winning region, there exists two output valuations for the same input valuation. In the specification of the voting contract given in Specification 7.1, for example, it is easy to forget the obligation that field `voters` should not change if `vote` was not called. This would result in two strategies for the case that method `close` is called: either set update $[\![\texttt{voters} \leftarrowtail \texttt{voters}]\!]$ or update $[\![\texttt{voters} \leftarrowtail \texttt{add(sender, voters)}]\!]$.

Deadlocks indicate that, at some point, there is an allowed combination of predicate evaluations such that no method can be called. For the deadlock detection, we require the developer to label predicate terms as *determined*. We call a predicate term determined if it has a fixed value at every step (until a method is called successfully). Some predicate terms can be syntactically recognized as determined, e.g., predicate terms that only have fields and constants, i.e., no inputs, as base terms. The value of such a predicate term cannot change until a method is called that changes the value of the fields. Predicate terms like `sender` ∈ `voters`, in contrast, are not determined as the evaluation always depends on the input `sender`.

As an example, we extend the voting specification such that the election can only close once a minimum number of votes has been cast. We use an additional field `numVotes`, which records the number of votes. The extended specification is depicted in Specification 7.2. In the specification, there are two determined predicate terms: `numVotes` > `cNum()` and `time` > `cTime()`. The first predicate term is determined as it does not contain any input variable. The second predicate term does contain an input variable (`time`), but as it is supposed to refer to the current time and since time is monotone, the predicate term cannot switch to *false* once it evaluates to *true*. If both predicate terms are labeled as determined by the developer, SCSYNT warns of a potential deadlock: in the initial state, there is no callable method if both `time` > `cTime()`

```
...
Fields: ..., numVotes
Constants: ..., cNum()

--- Additional Assumptions ---
□(vote ∧ ⊖ numVotes > cNum() → numVotes > cNum());

--- Additional Requirements ---
□(close → numVotes > cNum());

--- Additional Obligations ---
□(vote → ⟦numVotes ↢ addOne(numVotes)⟧);
□(⟐ close → ⟦numVotes ↢ numVotes⟧);
```

Specification 7.2: Extended voting specification.

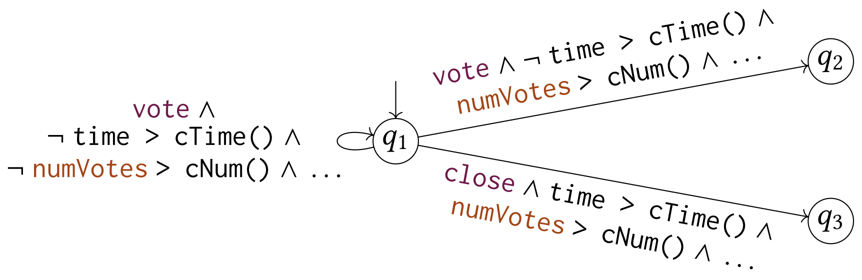

vote ∧ ¬ time > cTime() ∧ ¬ numVotes > cNum() ∧ ...
q1
vote ∧ ¬ time > cTime() ∧ numVotes > cNum() ∧ ...
q2
close ∧ time > cTime() ∧ numVotes > cNum() ∧ ...
q3

Figure 7.3: Deadlock in state machine of the extended voting specification.

and numVotes ≤ cNum() hold. Figure 7.3 depicts the part of the synthesized state machine that contains the deadlock. To prevent a deadlock, the specification can be adapted, e.g., by weakening the voting or closing condition. For now, this analysis can only warn of *potential* deadlocks. We labeled the time > cTime() predicate term as determined since it is determined if it is set to *true*. If it is *false*, then it is not determined, as it might switch to *true* as time progresses. Our analysis would warn of a potential deadlock also in a state that requires the term to be *false* to progress, as we do not distinguish for which values a predicate term is determined.

7.3 Specifying Smart Contracts with Parameterized PastTSL

We formally define TSL with parameters and demonstrate how to use pastTSL with parameters for the specification of smart contract control flows.

7.3.1 Parameterized TSL

Technically, parameterized TSL is TSL with infinitely many constants and cells. Let P be a set of parameters. We define the syntax of parameterized TSL with respect to P and a set $\mathbb{C}_P$ of parameterized cells. Each parameterized cell is of the form $\mathtt{c}(p_1, \ldots, p_m)$ with $p_1, \ldots, p_m \in P$. A parameterized TSL formula is a formula $\forall p_1, \ldots, \forall p_n.\, \psi$, where ψ is a TSL formula with cells from $\mathbb{C}_P$ and which allows parameters P as base terms in function terms and predicate terms. We require that the formula is closed, i.e., every parameter occurring in ψ must be bound in the quantifier prefix.

TSL formulas with parameters must be evaluated with respect to a domain for the parameters. Let $\mathbb{P}$ be such a (possibly infinite) set. To keep the definition of the semantics simple, we assume that $\mathbb{P}$ is a subset of the set of constants, i.e., 0-ary function terms. We use $\mu : P \to \mathbb{P}$ to denote a function that instantiates all parameters. For each parameter instantiation $\mu : P \to \mathbb{P}$, we further assume that $\mathtt{c}(\mu(p_1), \ldots, \mu(p_m)) \in \mathbb{C}$, i.e., that the instantiation of a parameterized cell refers to a normal cell. The set of cells $\mathbb{C}$ may be infinite. Given a parameterized TSL formula $\forall p_1, \ldots, \forall p_n.\, \psi$, let $\psi[\mu]$ be the formula obtained by replacing all parameters in ψ according to μ. As $\mathbb{P}$ is a subset of the set of constants and $\mathtt{c}(\mu(p_1), \ldots, \mu(p_m)) \in \mathbb{C}$, $\psi[\mu]$ is a TSL formula. Given a computation ς and an input stream ι, we define $\varsigma, \iota \models \forall p_1, \ldots, \forall p_n.\, \psi$ iff $\forall \mu : P \to \mathbb{P}.\, \varsigma, \iota \models \psi[\mu]$.

7.3.2 Parameterized Voting Specification

Using parameters, we can specify the voting example from last section more directly using a parameterized version of the `vote` method. We write `vote(m)` for a method call with parameter `m`, which is syntactic sugar for `isVote(methodinput, m)`. Parameters can either refer to a parameter of the method's signature or to the caller of the message. For the translation to Solidity, the developer needs to indicate for each parameter whether it should be instantiated with the caller or one of the method's arguments. In this example, `m` is the caller of the method. Compared with the voting specification from Section 7.1.1, we do not have to use a field that stores the voters in order to make sure that everybody only votes once. Instead, we simply state:

$$\forall \mathtt{m}.\, \square(\mathtt{vote}(\mathtt{m}) \to \circleddash \boxminus \neg \mathtt{vote}(\mathtt{m}))$$

The formula says that whenever `vote` is called by sender `m`, then `vote` has not been called in the past by that sender.

7.3.3 ERC20 Token System

As a second example we illustrate how specifications with parameters can be used to specify an ERC20 token system. We mainly follow the Open Zeppelin documentation [187] but also add additional functionality. An ERC20 token system provides a platform to transfer tokens between different accounts. The core contract consists of methods `transfer`, `transferFrom`, and `approve`. We do not model getters like `totalSupply` or `balanceOf` as they are not relevant for the temporal behavior of the contract. From a synthesis point-of-view, the challenge is that there is not only a global control flow but also one for each pair of accounts interacting with the contract. For example, if `n` wants to send tokens from `m`'s account to someone else, then `m` must have approved at least the requested amount to `n`. We tackle this challenge with a parameterized TSL specification using two parameters `m` and `n`. We use `m` for the contract from which tokens are subtracted and, whenever different from `m`, parameter `n` for the contract that initiates the transfer. In the following, we do not explicitly add the universal quantification as all parameters are always universally quantified. First, whenever either `transfer` or `transferFrom` is called, `m` must have a sufficient balance.

$$\square(\texttt{transfer(m)} \lor \texttt{transferFrom(m,n)} \rightarrow \texttt{sufficientFunds(m, arg@amount)})$$

We use `arg@` to label the current method's arguments that are not modeled as TSL parameters. We do not have to distinguish between calls to `transfer` with different values for `amount`, therefore we model the argument as normal TSL input. If `n` wants to transfer from `m`'s account, the amount must have been approved.

$$\square(\texttt{transferFrom(m,n)} \rightarrow \texttt{approved(m,n)} \geq \texttt{arg@amount})$$

The parameterized field `approved(m,n)` records the amount of tokens `m` allows `n` to spend in their name. It must be updated appropriately when `approve(m,n)` is called or when some of it is spent.

$$\square(\texttt{approve(m,n)} \rightarrow [\![\texttt{approved(m,n)} \leftarrowtail \texttt{arg@amount}]\!])$$

$$\square(\texttt{transferFrom(m,n)} \rightarrow [\![\texttt{approved(m,n)} \leftarrowtail \texttt{approved(m,n)} - \texttt{arg@amount}]\!])$$

$$\begin{aligned}&\square(\neg(\texttt{transferFrom(m,n)} \lor \texttt{approve(m,n)})\\&\quad \rightarrow [\![\texttt{approved(m,n)} \leftarrowtail \texttt{approved(m,n)}]\!])\end{aligned}$$

The Open Zeppelin ERC20 contract also offers various extensions of the core contract, one of which is the ability to pause all transfers by calling `pause` in case anything suspicious happens. `unpause` resumes all transfers. To illustrate how the use of parameters might lead to complex control flows, we additionally add method `localPause(m)` and `localUnpause(m)`. While `pause` stops *all* other method calls, `localPause(m)` should only stop transfers from `m`'s account. We specify that no method call can happen after

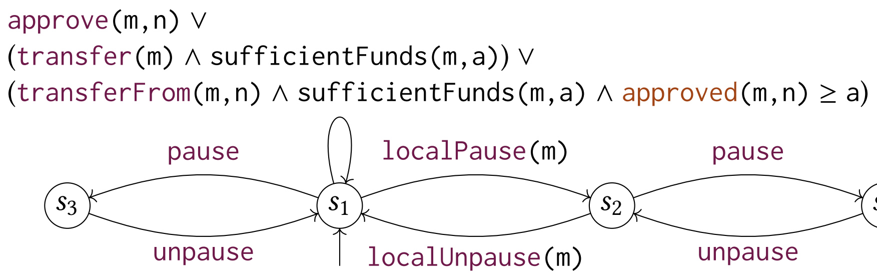

approve(m,n) ∨
(transfer(m) ∧ sufficientFunds(m,a)) ∨
(transferFrom(m,n) ∧ sufficientFunds(m,a) ∧ approved(m,n) ≥ a)
pause
localPause(m)
pause
s_3
s_1
s_2
s_4
unpause
localUnpause(m)
unpause

Figure 7.4: Synthesized state machine of the ERC20 token system with local pause function. Predicates that are not relevant for the transition and all field updates are omitted for readability. We also write a instead of `arg@amount`.

`pause` until `unpause` is called.

$$\square(\mathtt{transferFrom}(m,n) \lor \mathtt{transfer}(m) \lor \mathtt{approve}(m,n) \lor \mathtt{localPause}(m) \\ \lor \mathtt{localUnpause}(m) \\ \rightarrow (\neg\mathtt{pause}\ \mathcal{S}\ \mathtt{unpause}) \lor \boxminus \neg\mathtt{pause})$$

In contrast, `localPause(m)` only stops method calls related to `m`'s account; all others can continue.

$$\square(\mathtt{transferFrom}(m,n) \lor \mathtt{transfer}(m) \lor \mathtt{approve}(m,n) \\ \rightarrow (\neg\mathtt{localPause}(m)\ \mathcal{S}\ \mathtt{localUnpause}(m)) \lor \boxminus \neg\mathtt{localPause}(m))$$

Finally, we want that `pause` and `unpause` cannot be called twice without the respective other in between. Additionally, `unpause` cannot be called if `pause` has not been called at least once.

$$\square(\mathtt{unpause} \rightarrow \ominus(\neg\mathtt{unpause}\ \mathcal{S}\ \mathtt{pause}))$$

$$\square(\mathtt{pause} \rightarrow \ominus(\neg\mathtt{pause}\ \mathcal{S}\ \mathtt{unpause}) \lor \tilde{\ominus} \boxminus \neg\mathtt{pause})$$

The same two formulas are also required for `localPause` and `localUnpause`.

As parameters are universally quantified, the specification describes the same state machine for each of the infinitely many instantiations of the parameters. This state machine is depicted in Figure 7.4. The different instances are not independent, however. The contract can be in a different state for $(\mathtt{m} = 1, \mathtt{n} = 2)$ and $(\mathtt{m} = 3, \mathtt{n} = 4)$. But, if `pause` is called, both must move to a state when no transfer is possible. In the next section we discuss how we can represent the implied infinite-state machine in Solidity such that we ensure a correct interaction between the instances and keep the gas consumption at a reasonable level.

7.4 Parameterized PastTSL Synthesis

In this section, we describe how to extend the pastTSL synthesis described in [91] to specifications with universally quantified parameters. We first define parameterized LTL and how the TSL approximation in LTL [93] can be extended to approximating parameterized TSL by parameterized LTL. We subsequently state the problem definition in terms of parameterized LTL formulas, present the synthesis algorithm, and argue the correctness of our approach. Finally, we discuss the expressiveness limits of parameterized pastTSL for specifying the temporal control flow of smart contracts.

7.4.1 Parameterized LTL

Parameterized LTL has been studied in the settings parameterized verification [10] and parameterized synthesis [140]. We define it analogously to parameterized TSL and restrict ourselves to universally quantified formulas. Let AP be a set of *parameterized atomic propositions*, i.e., a set of propositions of the form $a(p_1, \ldots, p_m)$, where $p_1, \ldots, p_m \in P$. A parameterized LTL formula over AP is a closed formula $\forall p_1, \ldots, p_n.\, \psi$ such that $p_1, \ldots, p_n \in P$ and ψ is an LTL formula over AP. In this section, we usually denote the sequence $p_1, \ldots, p_m$ with some P_i, for which we also use set notation. We assume that every proposition occurs with only one sequence of parameters, i.e., there are no $a(P_i), a(P_j) \in AP$ with $P_i \neq P_j$. If a is not parameterized, $P_i = \emptyset$.

We evaluate parameterized LTL formulas with respect to the domain $\mathbb{P}$. Given a sequence of parameters $P_i = (p_1, \ldots, p_m)$ and an instantiation function $\mu : P \longrightarrow \mathbb{P}$, we write $P_i[\mu]$ for $(\mu(p_1), \ldots, \mu(p_m))$. For $a(P_i) \in AP$ we write, slightly abusing notation, $a \in AP$ and $a[\mu]$ for $a(P_i[\mu])$. As for parameterized TSL, $\varphi[\mu]$ is the formula where all atomic propositions are instantiated according to μ. Let $AP_{\mathbb{P}} = \{a[\mu] \mid a \in AP, \mu : P \longrightarrow \mathbb{P}\}$ be the set of all possible instantiations of AP. As there are no two $a(P_i), a(P_j) \in AP$ with $P_i \neq P_j$, for any $\alpha \in AP_{\mathbb{P}}$, there is exactly one a such that $a[\mu] = \alpha$ for some μ. Note that there might be multiple μ, μ' such that $a[\mu] = a[\mu']$. A trace t over $AP_{\mathbb{P}}$ satisfies a parameterized LTL formula $\forall p_1, \ldots, p_n.\, \psi$ if for all μ, it holds that $t \models \psi[\mu]$.

We extend the →LTL approximation of TSL to an approximation of parameterized TSL with parameterized LTL. We translate TSL predicate and update terms to LTL input and output propositions as described in [93] but list all parameters a term contains as part of the proposition. For example, `a(m) > b(n)` is translated to the parameterized proposition $c(m, n)$, where c is some suitable name for the predicate term. →page 80

7.4.2 Problem Definition

Let a parameterized pastLTL formula $\varphi = \forall p_1, \ldots, p_n.\, \psi$ be given, which is the approximation of a parameterized pastTSL formula specifying a smart contract. We denote the set of atomic propositions that correspond to some method call $\mathtt{f}(P_i)$ by $I_{call} \subseteq I$ and the set of output propositions that denote self-updates of fields $\mathtt{o}(P_i)$ by $O_{self} \subseteq O$.

Our aim is to synthesize a state machine that represents a smart contract from the parameterized LTL formula. We define when an infinite-state machine implements a smart contract with parameters.

Definition 7.1. Let $AP = I \cup O$ be a set of parameterized propositions and let $\mathbb{P}$ be a parameter domain. An infinite-state machine $\mathcal{M}$ over $2^{AP_{\mathbb{P}}}$ *implements a smart contract* if (i) for each transition $(s, A, s') \in \delta$, there is exactly one $\alpha \in A$ such that there is a μ and a $\mathtt{f}(P_i) \in I_{call}$ with $\alpha = \mathtt{f}(P_i)[\mu]$ and (ii) in every state $s \in S$, for every instance μ, and all inputs $A_I \subseteq I$, there exists a transition $(s, A, s') \in \delta$ such that $\{a[\mu] \mid a \in A_I\} \subseteq A$.

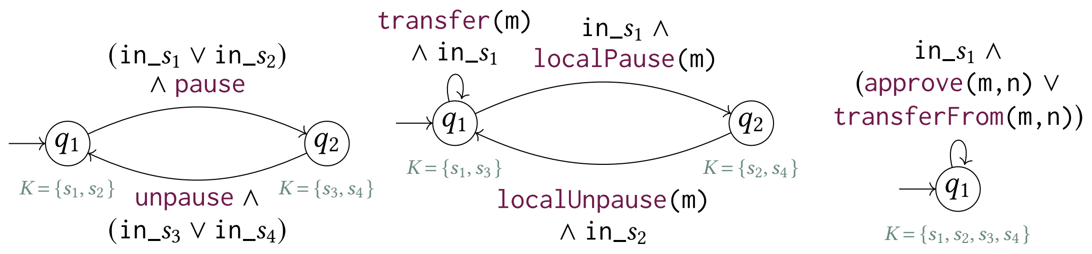
The definition requires that in each state of the state machine, every instance must have the possibility to make progress. In each transition, on the other hand, the state machine processes a method call for exactly one instance of the parameters. This corresponds to the intuition that only one method with certain arguments can be called at a time. This does not mean that only one instance makes progress per transition. For example, if `localPause(m)` is called with `m` = 1 in the ERC20 token system from Section 7.3.3, this affects all instances with $\mu(\mathtt{m}) = 1$. To project on steps relevant for an instance μ, we project the traces of $\mathcal{M}$ to steps that either include a method call to μ or a non-self-update of one of μ's fields. For $A \subseteq AP_{\mathbb{P}}$, we define A_μ as $\{\alpha \in A \mid \exists a \in AP.\, \alpha = a[\mu]\}$. Given $t \in \mathit{traces}(\mathcal{M})$, let $t' = (t[0])_\mu (t[1])_\mu \ldots$. Now, we define t_μ to be the trace obtained from t' by deleting all positions i such that $(t[i]_\mu)_{|O} \subseteq O_{self}$ and $\neg\exists \mathtt{f}(P_i) \in I_{call}.\, \mathtt{f}(P_i)[\mu] \in t'[i]$. Note that t_μ might be a finite trace even if t is infinite. Since t_μ only deletes steps from t that do not change the value of the cells, t_μ still constitutes a sound computation regarding the original pastTSL formula. We define $\mathit{traces}_\mu(\mathcal{M}) = \{t_\mu \mid t \in \mathit{traces}(\mathcal{M})\}$. Similarly, $\mathit{finTraces}_\mu(\mathcal{M}) = \{t_\mu \mid t \in \mathit{finTraces}(\mathcal{M})\}$.

Definition 7.2. A state machine implementing a smart contract *satisfies a pastLTL formula* $\varphi = \forall p_1., \ldots, \forall p_m.\, \psi$ if for all traces $t_\mu \in \mathit{traces}_\mu(\mathcal{M})$ and instances μ, we have $t_\mu \models \psi[\mu]$.

7.4.3 Synthesizing Smart Contracts with Parameters

To synthesize a state machine implementing a smart contract, we have to solve two challenges. First, the state machine needs to be finitely representable, which is a common challenge also faced in program synthesis or distributed reactive synthesis (see →Chapter 9). Second, we need to handle (in)dependencies between parameterized methods. Consider for example the ERC20 token system described in Section 7.3.3. Here, we have two parameters `m` and `n`. If method `localPause(m)` is called for some

→page 128



(a) System for ∅. (b) System for {m}. (c) System for {m, n}.

Figure 7.5: Synthesized state machines for all parameter sets with non-empty method set. For readability, we only display the knowledge states for each state and, for each transition, the method and the guards concerning the knowledge states; predicates and field updates are omitted.

address `m = 1`, the next state may not depend on predicates over parameter `n` like `approved(m,n) >= arg@amount`, as `n` is not assigned in the call to `localPause(m)`. While this is quite obvious from the programming perspective, it is not from the perspective of strategies represented as state machines: a strategy needs to know all input propositions in order to decide on what transition to take. We have to resolve this discrepancy between the different ways of thinking about a system, while also bearing in mind that the implementation in Solidity should be as gas efficient as possible.

Approach in a Nutshell. In a nutshell, we proceed as follows. First, we interpret the specification as being unquantified, i.e., we treat parameters as being part of the proposition's name. The synthesized finite-state machine then describes the behavior of a single instance of the parameters. The different instances depend on each other; therefore, it is not possible to just keep a duplicate of the state machine for every instance: again, if `localPause(m)` is called for `m` = `1`, this needs to be recorded in all state machines for instances that share the value of `m`. We would have to iterate over mappings to update all state machines of so-far observed instances, which would be very costly in Solidity.

To obtain a more feasible solution, we therefore describe an algorithm to split the synthesized state machine into multiple machines, one for each subset of parameters that occur in method call propositions. Together, these systems represent the full system. For the ERC20 example, we obtain the three systems depicted in Figure 7.5. Each node is labeled with a "knowledge" set K. If we follow a run of the original system in all subsystems, the knowledge states denote which state the original system might be in. $K = \{s_1, s_2\}$ in state q_1 of Figure 7.5a, for instance, means that the original system must be either in state s_1 or in state s_2. We furthermore add guards to the transitions, where in_s_1 means that the subsystem can only take the transition if it knows for sure that the full system is currently in state s_1. The systems share their knowledge about the state of the original system in a hierarchical structure. A subsystem for parameters P_i may share its knowledge only with systems for parameters $P_j \supseteq P_i$. We define an independence check that is successful if every subsystem comes at every state to a definite conclusion whether a transition can be taken.

In the following, we first describe two technical requirements that the synthesized finite-state machine has to satisfy in order to be split into multiple state machines. Then, we describe the splitting algorithm, the independence criterion, and how the distributed knowledge about the current state of the system can be put together again to construct an infinite-state machine. Finally, we argue that the approach is sound, i.e., if a system is decomposable according to the requirements and the independence criterion, then the implied infinite-state machine describes a smart contract that produces traces which satisfy the parameterized formula.

Technical Requirements. Let $\varphi = \forall p_1 \ldots \forall p_n. \psi$ and let $\mathcal{W}$ be the finite-state machine over AP that represents the winning region of ψ. Let $S_{\mathcal{W}}$ be the states of $\mathcal{W}$ and $\delta_{\mathcal{W}}$ its transition relation. We define two requirements on $\mathcal{W}$ which can be checked easily by inspecting all its transitions. The first requirement ensures that calls to a method parameterized with parameter sequence P_i only lead to field updates parameterized with the same parameter sequence.

Requirement 1 (Local Updates). *For every transition $(s, A, s') \in \delta_{\mathcal{W}}$, if $\mathtt{o}(P_i) \in A_{|O}$ and $\mathtt{o}(P_i) \notin O_{self}$, then there is a method call proposition $\mathtt{f}(P_i) \in A$.*

Second, whether a method can be called at a given state must not depend on predicates that are parameterized with parameters that are not included in the current method call. We formalize this by requiring that for every method call transition, there is another transition for the same method with a different valuation of such irrelevant input propositions.

Requirement 2 (Independence of Irrelevant Predicates). *For every $(s, A, s') \in \delta_{\mathcal{W}}$, if $\mathtt{f}(P_i) \in A$, then for every $a(P_j) \in I$ with $P_j \not\subseteq P_i$ and $a(P_j) \notin I_{call}$, there is a transition (s, A', s') with $a(P_j) \in A$ iff $a(P_j) \notin A'$ and $A_{|O} = A'_{|O}$.*

Splitting Algorithm. If $\mathcal{W}$ satisfies the above requirements, we proceed to split the state machine into multiple state machines, one for every subset P_i of parameters. Each $\mathcal{W}_{P_i}$ projects $\mathcal{W}$ to the method calls parameterized with parameters P_i. Furthermore, we label each state of every $\mathcal{W}_{P_i}$ with the current knowledge of the state machine regarding the global state of the contract. The algorithm to construct $\mathcal{W}_{P_i}$ combines several standard automata-theoretic concepts:

1. Introduce a new guard proposition $\mathtt{in\_}s$ for every state $s \in S_{\mathcal{W}}$ of $\mathcal{W}$. For every transition $(s, A, s') \in \delta_{\mathcal{W}}$, replace A with $A \cup \{\mathtt{in\_}s\}$.

2. Label all transitions $(s, A, s') \in \delta_{\mathcal{W}}$ for which there is no $\mathtt{f}(P_i) \in A$ with ϵ. The result is a nondeterministic →safety automaton with ϵ-edges. →page 19

3. $\mathcal{W}_{P_i}$ is obtained by determinizing the safety automaton using the standard subset construction. This removes all ϵ transitions. During the construction, we label each state with the subset of $S_{\mathcal{W}}$ it represents, these are the knowledge sets.

The subset construction employed in the algorithm above was initially defined for nondeterministic finite automata [196]. As safety automata are basically NFAs without accepting states, the construction can be straightforwardly applied to safety automata with ϵ-edges. We use S_i for the states of $\mathcal{W}_{P_i}$, δ_i for its transition relation, and $K_i : S_i \longrightarrow 2^{S_{\mathcal{W}}}$ for the knowledge sets. Note that every transition in $\mathcal{W}$ is labeled with exactly one method call proposition and is therefore present in exactly one $\mathcal{W}_{P_i}$. The following two propositions follow from the correctness of the standard subset construction for the determinization of finite automata. The first proposition states that the outgoing transitions of a state $s_i \in \mathcal{W}_{P_i}$ are exactly the outgoing transitions of all states $s \in K_i(s_i)$.

Proposition 7.1. *For every state $s_i \in S_i$, if $s \in K_i(s_i)$, then $(s, A, s') \in \delta_{\mathcal{W}}$ iff $(s_i, A \cup \{\mathtt{in\_}s\}, s_i') \in \delta_i$.*

The second one states that the knowledge labels in $\mathcal{W}_{P_i}$ are consistent with the transitions of $\mathcal{W}$.

Proposition 7.2. *Let $(s, A, s') \in \delta_{\mathcal{W}}$ with $\mathtt{f}(P_i) \in A$. Then, for every state $s_i \in S_i$ with $s \in K_i(s_i)$, and every transition $(s_i, A \cup \{\mathtt{in\_}s\}, s_i') \in \delta_i$, it holds that $s' \in K_i(s_i')$. Furthermore, for every s_j of $\mathcal{W}_{P_j}$ with $i \neq j$, if $s \in K_j(s_j)$, then $s' \in K_j(s_j)$.*

Check for Independence. We now define the check if transitions in $\mathcal{W}_{P_i}$ can be taken independently of the current state of all $\mathcal{W}_{P_j}$ with $P_j \not\subseteq P_i$. If the check is positive, we can implement the system efficiently in Solidity: when a method $\mathtt{f}(P_i)$ is called, we only need to make progress in system $\mathcal{W}_{P_i}$ and whether the transition can be taken only depends on the states of $\mathcal{W}_{P_j}$ with $P_j \subseteq P_i$ and on the values of the parameters in P_i. We describe the translation to Solidity in more detail in Section 7.5.

Let s_i and s_i' be states in $\mathcal{W}_{P_i}$ and $A \subseteq AP$. Let $G_{(s_i, A, s_i')} = \{s \mid (s_i, A \cup \{\mathtt{in\_}s\}, s_i') \in \delta_i\}$ be the set of all guard propositions that occur on transitions from s_i to s_i' with A. Let $P_{j_1}, \ldots P_{j_l}$ be the maximum set of parameter subsets such that $P_{j_k} \subseteq P_i$ for $1 \leq k \leq l$. A transition (s_i, A, s_i') is independent if for all states $s_{j_1}, \ldots, s_{j_l}$ with $s_{j_k} \in S_{j_k}$ either

(i) $K_i(s_i) \cap \bigcap_{1 \leq k \leq l} K_{j_k}(s_{j_k}) \subseteq G_{(s_i, A, s_i')}$ or

(ii) $(K_i(s_i) \cap \bigcap_{1 \leq k \leq l} K_{j_k}(s_{j_k})) \cap G_{(s_i, A, s_i')} = \emptyset$.

The check combines the knowledge of $\mathcal{W}_{P_i}$ in state s_i with the knowledge of each combination of states from $\mathcal{W}_{P_{j_1}}, \ldots, \mathcal{W}_{P_{j_l}}$. For each potential combination, it must be possible to determine whether transition (s_i, A, s_i') can be taken. If the first condition is satisfied, then the combined knowledge leads to the definite conclusion that $\mathcal{W}$ is currently in a state where an A-transition can be taken. If the second condition is satisfied, if definitely cannot be taken. If none of the two is satisfied, then the combined knowledge of P_i and all P_{j_k} is not sufficient to reach a definite answer. Note that some state combinations $s_i, s_{j_1}, \ldots, s_{j_l}$ might be impossible to reach. But then, we have that $K_i(s_i) \cap \bigcap_{1 \le k \le l} K_{j_k}(s_{j_k}) = \emptyset$ and the second condition is satisfied. The check is successful if all transitions (s_i, A, s_i') in all δ_i are independent.

Construction of the Infinite-state Machine. If the above check is successful, all $\mathcal{W}_{P_i}$ together represent an infinite-state machine $\mathcal{M}$ that implements a smart contract. We construct $\mathcal{M}$ as follows. A state in $\mathcal{M}$ is a collection of $n = |2^P|$ functions $f_1, \ldots, f_n$, where $f_i : \mathbb{P}^m \to S_i$ if $P_i = (p_{i_1}, \ldots, p_{i_m})$. Each f_i indicates in which state of $\mathcal{W}_{P_i}$ instance μ currently is. The initial state is the collection of functions that all map to the initial states of their respective $\mathcal{W}_{P_i}$. For every state $s = (f_1, \ldots, f_n)$ of $\mathcal{M}$, every $P_i \subseteq P$, and every instance μ, we add a transition where $P_i[\mu]$ makes progress and all other instances stay idle. Let $f_i(P_i[\mu]) = s_i$, $s_i' \in S_i$, $A \subseteq AP$, and $G_{(s_i,A,s_i')} = \{s \mid (s_i, A \cup \{\texttt{in\_}s\}, s_i') \in \delta_i\}$. Let $P_{j_1}, \ldots P_{j_l}$ be all subsets of P_i. If $K_i(s_i) \cap \bigcap_{1 \le k \le l} K_{j_k}(f_{j_k}(P_{j_k}[\mu])) \subseteq G_{(s_i,A,s_i')}$, we add the transition (s, A', s') to $\mathcal{M}$, where A' and s' are defined as follows.

$$
\begin{aligned}
A' &= \{a[\mu] \mid a \in A\} \cup \{o[\mu'] \mid o \in O_{self}, o[\mu'] \neq o[\mu]\} \\
s' &= (f_1, \ldots, f_i[P_i[\mu] \mapsto s_i'], \ldots, f_n)
\end{aligned}
$$

The label A' sets all propositions of instance μ as in A and sets all other input propositions to *false*. Of all other outputs propositions, it only sets those denoting self-updates to *true*.

We observe that, as $\mathcal{W}$ describes a collection of strategies for ψ, $\mathcal{M}$ describes a smart contract as defined in Definition 7.1. It remains to prove that the traces of $\mathcal{M}$ satisfy φ according to Definition 7.2. Most of the work is done in the following lemma. We define $\mathcal{W}_\mu$ as the state machine that replaces the transition labels of $\mathcal{W}$ with their instantiations according to μ, i.e., if $(s, A, s') \in \mathcal{W}$, then $(s, A[\mu], s') \in \mathcal{W}_\mu$.

Lemma 7.3. *For every μ, $traces_\mu(\mathcal{M}) \cup finTraces_\mu(\mathcal{M}) = traces(\mathcal{W}_\mu) \cup finTraces(\mathcal{W}_\mu)$.*

Proof. We show that every (finite or infinite) run $r_\mathcal{M}$ of $\mathcal{M}$ can be matched with a (finite or infinite) run $r_{\mathcal{W}_\mu}$ of $\mathcal{W}_\mu$ (and vice versa) such that $trace_\mu(r_\mathcal{M}) = trace(r_{\mathcal{W}_\mu})$. For the first direction of the equality, we show that every transition of $r_\mathcal{M}$ can be either matched with a transition in $\mathcal{W}_\mu$ or constitutes a step which is removed in the trace $t_\mu = trace(r_\mathcal{M})_\mu$. For the other direction, we show that every transition of

$r_{\mathcal{W}_\mu}$ can be matched with a transition in $\mathcal{M}$. Assume $\mathcal{M}$ is currently in state $s = (f_1, \ldots, f_n)$ of $r_\mathcal{M}$ and $\mathcal{W}_\mu$ is in state $s_\mathcal{W}$ of $r_{\mathcal{W}_\mu}$. We keep the invariant that $s_\mathcal{W} \in K_\mu = \bigcap_{1 \le j \le n} K_j(f_j(P_j[\mu]))$.

- For the first direction, assume that the next transition of $\mathcal{M}$ is (s, A, s') with $s' = (f'_1, \ldots, f'_n)$. Let $B \subseteq AP$ be such that $B[\mu] = A_\mu$. By construction of $\mathcal{M}$, there is an instance μ' and parameter subset P_i such that there is exactly one $\mathtt{f}(P_i[\mu']) \in A$. Let $B' \subseteq AP$ be such that $B'[\mu'] = A_{\mu'}$. We distinguish two cases.
  - Assume $P_i[\mu] = P_i[\mu']$. Let $s_i = f_i(P_i[\mu'])$ and $s'_i = f'_i(P_i[\mu'])$. Let $G_{(s_i, B', s'_i)} = \{s \mid (s_i, B' \cup \{\mathtt{in\_}s\}, s'_i) \in \delta_i\}$. Let $P_{j_1}, \ldots P_{j_l}$ be all subsets of P_i and $K = K_i(s_i) \cap \bigcap_{1 \le k \le l} K_{j_k}(f_{j_k}(P_{j_k}[\mu']))$. By definition of the independence check employed to construct $\mathcal{M}$, $K \subseteq G_{(s_i, B', s'_i)}$. As $P_i[\mu] = P_i[\mu']$, we have that $K_\mu \subseteq K$ and thus also $K_\mu \subseteq G_{(s_i, B', s'_i)}$. Therefore, by the invariant, $s_\mathcal{W} \in G_{(s_i, B', s'_i)}$ and there is a transition $(s_\mathcal{W}, B', s'_\mathcal{W})$ in $\mathcal{W}$.
    Now, if $\mu' = \mu$, there is a transition $(s_\mathcal{W}, B[\mu], s'_\mathcal{W})$ in $\mathcal{W}_\mu$, what shows that the step in $\mathcal{M}$ can be mirrored in $\mathcal{W}_\mu$. Otherwise, if $\mu' \neq \mu$, B and B' agree on the propositions parameterized with P_i. By construction of $\mathcal{M}$, B has all inputs not parameterized with P_i set to *false* and only self-updates of cells not parameterized with P_i set to *true*. By Requirements 1 and 2, since there is a transition $(s_\mathcal{W}, B', s'_\mathcal{W})$ in $\mathcal{W}$, there is also a transition $(s_\mathcal{W}, B, s'_\mathcal{W})$ in $\mathcal{W}$ and therefore a transition $(s_\mathcal{W}, B[\mu], s'_\mathcal{W})$ in $\mathcal{W}_\mu$.
    All that remains to show is that the invariant is preserved by both transitions $(s_\mathcal{W}, B[\mu], s'_\mathcal{W})$ and $(s_\mathcal{W}, B'[\mu'], s'_\mathcal{W})$. That means that we need to show that $s'_\mathcal{W} \in K'_\mu$ for $K'_\mu = \bigcap_{1 \le j \le n} K_j(f'_j(P_j[\mu]))$ in either case. Since $s_\mathcal{W} \in K_\mu$, we have that for every j, $s_\mathcal{W} \in K_j(f_j(P_j[\mu]))$. Furthermore, $\mathtt{f}(P_i) \in B$ and $\mathtt{f}(P_i) \in B'$. For $j \neq i$ we have that $f'_j = f_j$ and, therefore, by Proposition 7.2, $s'_\mathcal{W} \in K_j(f'_j(P_j[\mu]))$. Furthermore, $K_\mu \subseteq G_{(s_i, B', s'_i)}$ and $(s_i, C \cup \{\mathtt{in\_}s_\mathcal{W}\}, s'_i) \in \delta_i$, for both $C = B$ and $C = B'$. Therefore, again by Proposition 7.2, $s'_\mathcal{W} \in f_i[P_i[\mu] \mapsto s'_i]$. Thus, $s'_\mathcal{W} \in K_j(f_j(P_j[\mu]))$ for all j and the invariant holds for $s'_\mathcal{W}$.
  - Assume $P_i[\mu] \neq P_i[\mu']$. By construction of $\mathcal{M}$, there is no $\mathtt{g}(P_j) \in B$. As $\mathcal{W}$ satisfies Requirement 1, so does every $\mathcal{W}_{P_i}$. Together with how $\mathcal{M}$ is constructed, it follows that $B_{|O} \subseteq O_{self}$. Thus, by definition of t_μ, the transition will not appear in t_μ and $\mathcal{W}_\mu$ thus stays in its current state. In remains to show that the invariant is maintained by the transition in $\mathcal{M}$. Since $P_i[\mu] \neq P_i[\mu']$, we have by definition of $\mathcal{M}$ that $f_j(P_j[\mu]) = f'_j(P_j[\mu])$ for all j, and therefore the invariant is maintained.
- For the second direction, assume that the next transition of $\mathcal{W}_\mu$ is $(s_\mathcal{W}, B[\mu], s'_\mathcal{W})$ for some $B \subseteq AP$. By construction of $\mathcal{W}$, there is a parameter subset P_i such that

some $\texttt{f}(P_i) \in B$. By our invariant, $s_{\mathcal{W}} \in K_i(s_i)$, therefore, by Proposition 7.1, there is a transition $(s_i, B \cup \{\texttt{in\_}s_{\mathcal{W}}\}, s'_i) \in \delta_i$. By construction of $\mathcal{M}$, there is a transition from $(f_1, \ldots, f_n)$ to $(f'_1, \ldots, f'_n)$ such that $f'_i(P_i[\mu]) = s'_i$. Furthermore, the transition is labeled with A such that $A_\mu = B[\mu]$.

It remains to show that the invariant is maintained by the transition. For $j \neq i$, $f'_j = f_j$ and therefore, by Proposition 7.2, $s'_{\mathcal{W}} \in K_j(f_j(P_j[\mu]))$. Furthermore, since $(s_i, B \cup \{\texttt{in\_}s_{\mathcal{W}}\}, s'_i) \in \delta_i$, again by Proposition 7.2, it holds that $s'_{\mathcal{W}} \in K_i(s'_i)$ and therefore, by definition of $f'_i = f_i[P_i[\mu] \mapsto s'_i]$, $s'_{\mathcal{W}} \in K_i(f'_i(P_i[\mu]))$. Therefore, the invariant is maintained for the transition. □

Not every infinite run of $\mathcal{M}$ corresponds to an infinite run in $\mathcal{W}_\mu$ for every μ. Therefore, the above lemma only holds for the union of finite and infinite traces. However, as we work with past-time logics, a trace satisfies a formula iff all its prefixes satisfy the formula. Thus, a state machine $\mathcal{M}$ satisfies a pastLTL formula ψ iff for every $t \in \mathit{traces}(\mathcal{M}) \cup \mathit{finTraces}(\mathcal{M})$, it holds that $t \models \psi$. From the above lemma we therefore directly obtain the desired correctness result for $\mathcal{M}$.

Theorem 7.4. *Let $\varphi = \forall p_1, \ldots p_m. \psi$ be a parameterized pastLTL formula obtained by approximating a parameterized pastTSL formula ρ. If $\mathcal{W} \models \psi$, $\mathcal{W}$ satisfies Requirements 1 and 2, and can be split into $\mathcal{W}_{P_1}, \ldots, \mathcal{W}_{P_n}$ such that the check for independence is successful, then $\mathcal{M} \models \varphi$.*

Since all parameters are universally quantified, it follows from the soundness of the LTL approximation of TSL formulas [93] that the infinite-state machine $\mathcal{M}$ then also satisfies the original pastTSL formula ρ.

7.4.4 Limitations in Expressiveness

Most of the limitations of our approach originate from the trade-off between the expressiveness of our specification language and the performance of the tool SCSynt. One way to increase the expressiveness of the logic would be to combine pastTSL with domain-specific reasoning. There have been approaches to extend TSL with theories [90] and to include SMT solving into the synthesis loop [171]. The latter has only been successful for small specifications, however, as it needs an increasing number of refinement loops for larger specifications. A very recent approach combines TSL with SyGuS to automatically generate assumptions that make the specification realizable [43]. This approach might be more suitable for our setting, since we rely on fast synthesis to give the developer feedback on free choices and deadlocks.

Regarding the extension of pastTSL with parameters, the major limitation is that we currently cannot handle existential quantifiers. In the example of the ERC20 contract, this forbids us to use a field `funds(m)` to store the balance of all users of the contract. If we were to try, we could use an additional parameter r for the recipient of the tokens and state the following.

$$\begin{aligned}\forall \mathtt{m}, \mathtt{n}, \mathtt{r}. \square(&\mathtt{transferFrom}(\mathtt{m},\mathtt{n},\mathtt{r}) \vee \mathtt{transfer}(\mathtt{m},\mathtt{r}) \\ &\rightarrow [\![\mathtt{funds}(\mathtt{m}) \leftarrowtail \mathtt{funds}(\mathtt{m}) - \mathtt{arg@amount}]\!] \\ &\quad \wedge [\![\mathtt{funds}(\mathtt{r}) \leftarrowtail \mathtt{funds}(\mathtt{r}) + \mathtt{arg@amount}]\!])\end{aligned}$$

However, for completeness, we would have to specify that the funds field does not spuriously increase, which would require existential quantifiers.

$$\begin{aligned}\forall \mathtt{r}. \square(&[\![\mathtt{funds}(\mathtt{r}) \leftarrowtail \mathtt{funds}(\mathtt{r}) + \mathtt{arg@amount}]\!] \\ &\rightarrow \exists \mathtt{m}. \exists \mathtt{n}. \mathtt{transferFrom}(\mathtt{m},\mathtt{n},\mathtt{r}) \vee \mathtt{transfer}(\mathtt{m},\mathtt{r}))\end{aligned}$$

A similar limitation stems from Requirement 1, which requires that a field parameterized with set P_i can only be updated by a method which is also parameterized with P_i. As for existential quantifiers, we would otherwise not be able to distinguish spurious updates from intended updates of fields. While it might be hard to extend the approach with arbitrary existential quantification, it should be possible for future work to include existential quantification that prevents spurious updates. One could, for example, define some sort of "lazy synthesis", which only does a non-self-update when necessary.

7.5 Implementation and Evaluation

In this section, we describe how we extended SCSynt with the features described in this chapter. We furthermore report on our evaluation of the extended variant of SCSynt with regard to size and gas consumption of the generated contracts.

7.5.1 Implementation

We integrated the analysis for specification errors and the handling of parameters as part of the toolchain of SCSynt, which is mostly implemented in Python. Given a parameterized pastTSL specification, we interpret the formula as unparameterized, and let SCSynt compute its winning region by solving the safety game of the safety automaton. We analyze the winning region for free choices and, if the user provides determined predicates, for potential deadlocks. Finally, if the specification is parameterized, we split the system as described in Section 7.4.

The general schema of the Solidity code for parameterized specifications can be found in Solidity Code 7.2. We define enums for the states of all state machines (l. 2) and store the initial and the current state of each system (l. 6, 7). As the current state of a

```
1  contract <contract_name> {
2    enum State<P_i> { s1, ..., sk_{P_i} } // state machine states
3    enum KState { s1,..., sk_K } // knowledge labels
4    <type> private kMap<P_i>;  // sharing of knowledge
5    // initial and current states
6    State<P> private init<P_i> = State<P_i>.<initial state>;
7    <type> private currState<P_i>;
8    // constants
9    <type> private immutable <c_name>;
10   <type> private immutable <c_name> = <definition>;
11   // fields
12   <type> <access_modifier> <f_name>;
13   address private owner;
14   bool inMethod = false;
15   constructor(<args>) {
16     <init. of knowledge maps>
17     owner = msg.sender;
18     <c_name> = <definition>; // init. of constants
19   }
20   function m_i(<parameters>,<args>) public <is_payable> {
21     require(! inMethod);
22     inMethod = true;
23     <state_machine_logic_for_m_i>
24     inMethod = false;
25   }
26 }
```

Solidity Code 7.2: Skeleton of generated Solidity code. $\mathsf{k}_{\mathsf{P}_i}$ is the number of states in the state machine for P_i, k_K is the number of states in the full state machine.

state machine may differ for different instantiations of the parameters, `currState<P_i>` is a mapping that maps the instance to a state. We define a further enum for the states of the original system, which we use to denote the knowledge of the states (l. 3, see Section 7.4). For every two parameter sets, we define and initialize knowledge maps (l. 4,16) that implement the sharing of knowledge for pairs of states. The knowledge maps are nested mappings, which take a state from each of the two parameter sets and a state from the original system and return whether the original state is in the shared knowledge of the two parameter sets.

The rest of the code is similar to code produced for non-parameterized specifications. We declare all constants and fields that appear in the specification as well as a variable storing the owner's address (l. 8-13). Constants are labeled immutable and can either be assigned a predefined value provided by the developer (l. 10), or they can be initialized in the constructor (l. 9,18). The transition system logic is implemented in each of the methods of the contract. The parameters of a method are included as arguments or replaced with the corresponding fixed interpretation (e.g, if `m` is always `msg.sender`). This information needs to be provided by the developer as part of a signature that also defines the type of the methods and fields. As described in Section 7.1, the code also includes a flag `inMethod` (l. 14) that enforces the assumption that method calls are atomic and protects against reentrancy attacks.

7.5.2 Evaluation

We evaluate SCSynt with regard to the conciseness of the generated Solidity contracts. We synthesized ten different smart contracts, for some we needed specifications with parameters, for others we did not. The specifications without parameters stem from [91] and are not a contribution of this thesis, only their evaluation regarding conciseness is. The average synthesis time for the full tool chain was 2 seconds. The largest specification took 12 seconds to synthesize. The runtime of our extensions (the analysis and the splitting of the state machine for parameterized specifications) is negligible as they are performed on the minimized transition system, which has less then 10 states for all our specifications. We compare the generated Solidity code to handwritten contracts found in the literature in terms of both size and average gas consumption. Since the reference contracts do not provide any formal guarantees, we further compare SCSynt's Solidity code to formally correct contracts generated by two other approaches. First, we briefly describe the contracts we use as benchmarks.

Benchmark Contracts. Our evaluation includes a range of typical smart contracts that have a non-trivial underlying state machine. For each benchmark, we indicate whether the contract is parameterized and against which other implementation we compare the code produced by SCSynt.

- *ERC20 Token System* (parameterized, reference implementation: [186]). This is a typical ERC20 token system following the Open Zeppelin documentation [187]. The specification is similar to the one presented in Section 7.3.3, but does not contain the parameterized methods for pausing and unpausing a contract.
- *ERC20 Token System Extended* (parameterized). This is the contract described in Section 7.3.3 that extends the classical ERC20 token system with different methods for pausing.
- *Coin Toss* (not parameterized, reference implementation: [193]). Following the description of [230], this contracts implements the toss of a coin. Two parties bet on which side the coin lands. The first party commits to a side and places a wager. Then, the second party accepts the bet and deposits at least the same amount. The winner receives both wagers.

- *Voting* (parameterized, reference implementation: [104]). This is the parameterized version of the voting contract described in Section 7.1.1 and Section 7.3.2.

- *Asset Transfer* (not parameterized, reference implementation: [229]). This contract describes the safe transfer of an asset between two parties. Our specification follows the description of the state machine in Microsoft's Azure Blockchain Workbench [230]. Similar versions of the contract are described in [227, 66].

- *Blinded Auction* (not parameterized, reference implementation: [99]). This is a typical auction protocol with the special feature that all bids are hashed such that the current highest bid is not known while the auction is still ongoing. Our specification follows the description in [174].

- *Crowd Funding* (not parameterized, reference implementation: [194]). In a crowd funding protocol, users can send coins to fund a project. If the project is not funded before a time limit is reached, all users can reclaim their contribution. Otherwise, the owner receives the stock. Our specification follows the description in [216].

- *NFT Auction* (parameterized, reference implementation: [13]). This is a comparatively large contract from Avolabs that implements an NFT auction combined with a buy-now feature. The original implementation has over 1400 lines of code. We specified the main requirements on the control flow as described in the `README` of Avolabs' GitHub.

- *Simple Auction* (not parameterized, reference implementation: [103]). This is a simple version of an auction, similar to the blinded auction but without the hashing of the bids. A description can be found in [100].

- *Ticket System* (not parameterized, reference implementation written by the authors). This contracts describes a small ticket system. Users can by tickets for a specific amount of time. As long as the sale is still open, they can also return their tickets and claim a refund.

We evaluate SCSynt in two categories: comparison to handwritten contracts without formal guarantees on the temporal control flow and comparison to automatically generated contracts with formal guarantees.

Comparison to Handwritten Contracts. We compare the Solidity code generated by SCSynt to handwritten contracts for the same contracts found in the literature but without formal guarantees. We compare the contracts in their size, i.e., the number of lines of Solidity code, and in their average gas consumption. For a fair comparison, we

Table 7.1: Comparison in terms of contract size and average gas consumption of the contracts generated by SCSynt to the handwritten reference contracts (Ref.). The gas consumption is the sum of the average gas consumption of the deployment and all methods.

	`ERC20`		`ERC20 Ext.`		`Coin Toss`		`Voting`	
	SCSynt	**Ref.**	**SCSynt**	**Ref.**	**SCSynt**	**Ref.**	**SCSynt**	**Ref.**
#Lines	100	62	124	–	65	32	69	37
Gas	1837260	1076141	2390510	–	1092107	760954	1043467	769302

	`Asset Transfer`		`Blinded Auction`		`Crowd Funding`	
	SCSynt	**Ref.**	**SCSynt**	**Ref.**	**SCSynt**	**Ref.**
#Lines	74	62	107	91	55	24
Gas	1354451	998507	1513171	1484856	816614	455736

	`NFT Auction`		`Simple Auction`		`Ticket System`	
	SCSynt	**Ref.**	**SCSynt**	**Ref.**	**SCSynt**	**Ref.**
#Lines	300	457	55	35	46	21
Gas	4831300	3307183	759418	712151	647039	461578

extend our code with the implementation of the contract beyond the specified control flow. The resulting contracts thus implement exactly the same features. We measured the gas consumption using the Truffle Suite [218]. The results are shown in Table 7.1. We see that SCSynt generates significantly shorter Solidity code for the NFT auction protocol. This is mostly due to code duplication at various locations in the handwritten contract which is needed to structure the code and to make it readable. Since SCSynt produces code that, by construction, adheres to the specification and is thus correct, the code duplication is not necessary in our code; resulting in 35% fewer lines of code. For the other contracts, SCSynt generates longer Solidity code. It seems that SCSynt has an advantage in terms of code length for more complicated and thus generally longer smart contracts.

In terms of average gas consumption, the handwritten contracts have an advantage over SCSynt's code. On average, our synthesized code consumes 40% more gas than the handwritten, non-verified code. Approximately one third of the overhead stems from SCSynt's capability of preventing reentrancy attacks (see Section 7.1.1): without the additional code for preventing the attacks, SCSynt's code consumes only 25% more gas than the handwritten contracts. When distinguishing between contracts obtained from parameterized specifications and specifications without parameters, we see that the average gas overhead for parameterized contracts is 50.6% as compared to 34.5% for non-parameterized contracts. Considering that the implementation of the hierarchical systems and the sharing of knowledge needs some machinery like additional enums and mappings, this difference is not huge and was to be expected.

Comparison with Contracts with Formal Guarantees. There are no other tools available which synthesize the control flow of a smart contract including guards on the inputs and the control flow of the contract's fields. As a remedy, we consider two other approaches that generate Solidity code and provide formal guarantees on the control flow. The first approach automatically synthesizes a state machine from LTL specifications which is then translated to Solidity code [216]. It cannot express access rights, e.g., that only the owner can call certain methods. The second approach, the VeriSolid framework [174, 175], generates Solidity code from a handwritten state machine. It cannot express access rights either.

Both approaches provide Solidity code for the blinded auction benchmark. Therefore, we compare the code generated by SCSynt to their contracts. Both the synthesized state machine of the first approach and the manually defined state machine of the VeriSolid framework are very similar to ours except for the access rights which they lack. We observe that the Solidity code generated by the LTL synthesis approach apparently misses guards regarding the time limits. We adapted their code to match their state machine. The resulting code is 12% shorter than the contract generated with SCSynt and consumes 2% less gas. Since the LTL synthesis approach does not prevent reentrancy attacks, we further compare the approaches when disabling SCSynt's reentrancy prevention. Then, the generated contracts are of similar length and SCSynt's contract consumes 11% less gas on average. The VeriSolid framework cannot handle that the same method can be invoked in several states; in that case, the method is duplicated, one copy for each state in which it can be invoked. In the blinded auction contract, this only occurs for a single method. Thus, their code size for the blinded auction is only slightly affected by the code duplication. SCSynt's contract consumes 12% more gas on average. When disabling SCSynt's reentrancy prevention, the difference in the average gas consumption of both contracts is negligible: SCSynt's contract consumes 0.3% more gas on average.

Chapter 8

Smart Contract Synthesis Modulo Hyperproperties

In the previous chapter, we synthesized smart contracts from trace properties that describe their functional behavior. Smart contracts often additionally need to satisfy hyperproperties, however. Existing work on hyperproperties in smart contracts focuses on verifying concrete information flow policies such as integrity [41, 40] or effective callback freedom [118, 6]. Hyperproperties are not limited to information flow policies, though. To establish the users' trust in a contract, one has to additionally show that a contract satisfies hyperproperties such as robustness and symmetry. In a voting contract, for example, all candidates should be treated symmetrically and a vote from the owner of the contract should not count more than any other vote. In this chapter, we develop two variants of a novel temporal hyperlogic, HyperTSL, to express the wide range of hyperproperties relevant in the context of smart contracts. We then show how to synthesize smart contracts from specifications given in HyperTSL. We approach the task from two angles. First, we investigate the general synthesis problem of the logic and show how to approximate the in general undecidable problem. Second, we propose a two-step approach to include HyperTSL properties in the synthesis of smart contracts from pastTSL specifications as described in last chapter.

We develop two extensions of TSL to hyperproperties: HyperTSL and HyperTSL$_{\mathrm{rel}}$. We define HyperTSL as a conservative extension of TSL. It adds quantification over multiple executions; predicates, functions, and updates then refer to one of the quantified executions. In an election with two candidates A and B, this makes it possible to state that two traces have the same winner A if they agree on the votes for A:

$$\forall \pi, \pi'. \square(\mathtt{voteA}_{\pi} \leftrightarrow \mathtt{voteA}_{\pi'}) \rightarrow \square(⟦\mathtt{winner} \leftarrowtail \mathtt{A()}⟧_{\pi} \leftrightarrow ⟦\mathtt{winner} \leftarrowtail \mathtt{A()}⟧_{\pi'})$$

The second hyperlogic, HyperTSL$_{\mathrm{rel}}$, can relate multiple executions within a predicate. Here, we can state that the winners on two executions are always the same as long as the votes are always the same:

$$\forall \pi, \pi'. \Box(\texttt{vote}_\pi = \texttt{vote}_{\pi'}) \rightarrow \Box(\texttt{winner}_\pi = \texttt{winner}_{\pi'})$$

Note that the = predicate ranges over two different execution variables π and π'. In TSL, all functions and predicates are uninterpreted, but = would probably be implemented as actual equality.

Our goal is to synthesize smart contract control flow graphs that satisfy HyperTSL and $\text{HyperTSL}_{\text{rel}}$ specifications. Inherited from TSL, the synthesis problem is undecidable already for a single universal or existential quantifier. We show, however, that the $\forall^*$ fragment of HyperTSL can be approximated in $\forall^*$ HyperLTL, for which there exists a bounded synthesis approach implemented in BoSyHyper [86]. For the $\exists^*$ fragment of HyperTSL, we present an approximation based on LTL satisfiability checking.

As a step towards actual smart contract synthesis, we build on the synthesis from pastTSL from [91] as described in →Section 7.1. We present a two-step approach to synthesize a contract that adheres to HyperTSL specifications. First, we check if the combination of a given TSL specification with a $\forall^*$ HyperTSL specification lets the hyperproperty "collapse" to a simple trace property. We show that the check is undecidable in general, but can to some extend be approximated in a fragment of HyperLTL for which satisfiability checking is decidable. If this is not the case, we synthesize the winning regions of the pastTSL specification and prune the system to find a strategy that implements a $\forall^*$ HyperTSL property. Our implementation shows that we can automatically construct a voting contract which satisfies several hyperproperties. →page 84

Outline. In Section 8.1, we introduce HyperTSL and $\text{HyperTSL}_{\text{rel}}$. We continue by showing how hyperproperties in smart contracts can be specified in the logics in Section 8.2. In Section 8.3, we discuss the general HyperTSL synthesis problem and present approximations in LTL and HyperLTL. Finally, in Section 8.4 and Section 8.5, we show how to integrate HyperTSL specifications in the contract synthesis from pastTSL.

Publications. This chapter presents the contents of the following publication.

[58] Norine Coenen, Bernd Finkbeiner, Jana Hofmann, and Julia Tillman. **Smart Contract Synthesis Modulo Hyperproperties**. *To appear at the 36th IEEE Computer Security Foundations Symposium (CSF 2023).*

Parts of the above paper build on the results of Julia Tillman's Bachelor thesis at Saarland University in 2020, which the author of this thesis supervised together with Norine Coenen. The Bachelor's thesis contains earlier versions of the logics HyperTSL and $\text{HyperTSL}_{\text{rel}}$ as well as the approximation of $\forall^*$ HyperTSL synthesis via HyperLTL.

8.1 TSL for Hyperproperties

We extend TSL with execution quantifiers to allow reasoning about hyperproperties for infinite-state systems. Compared with other temporal logics describing trace properties, there is not a single straight-forward way to lift TSL to a hyperlogic.

Function and Predicate Interpretations. There are two options to deal with function and predicate interpretations. The first is to interpret functions and predicates differently across multiple executions. This would make sense, for example, to model multiple components of a system. Most of the hyperproperties, however, concern the behavior of the same system when executed several times. Functions and predicates like equality and addition mostly stay the same across several executions. We therefore decide to quantify the function and predicate interpretation *before* quantifying executions.

Domain of Updates and Predicates. Another question is whether predicates, functions, and update terms may contain terms evaluated on different executions. In the case of update terms, the answer should probably be no: an update of the form $[\![\mathsf{c}_\pi \leftarrowtail f(\mathsf{c}_{\pi'})]\!]$ would mean that the system could access the value of a cell at the same point in time but on a different execution. This does not seem realistic for a real system, which would have to store all possible values a cell c could hold at some point in time. Additionally, the execution chosen for π' differs with every quantifier evaluation. Predicates ranging over multiple executions, on the other hand, would be useful. One could state a stronger version of observational determinism, namely "if two traces agree on the value of input x, then also cell c is always updated to the same term", written as $\square(x_\pi = x_{\pi'}) \rightarrow \square([\![\mathsf{c} \leftarrowtail f(x)]\!]_\pi \leftrightarrow [\![\mathsf{c} \leftarrowtail f(x)]\!]_{\pi'})$. We therefore define two logics: HyperTSL, in which we allow predicates to range only over the *same* execution, and HyperTSL$_{\text{rel}}$, which relates multiple executions using predicates.

8.1.1 HyperTSL

We define HyperTSL as an extension of →TSL. The definition of function terms τ^f and predicate terms τ^p are the same as in TSL. The syntax of HyperTSL is given as follows. Let V_π be a set of trace variables. →page 77

$$\begin{aligned} \varphi &::= \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \psi \\ \psi &::= \neg\psi \mid \psi \wedge \psi \mid \bigcirc\psi \mid \psi\, \mathcal{U}\, \psi \mid \tau^p_\pi \mid [\![\mathsf{c} \leftarrowtail \tau^f]\!]_\pi \end{aligned}$$

where $\pi \in V_\pi$, $\mathsf{c} \in \mathbb{C}$, $\tau^p \in \mathcal{T}_P$, and $\tau^f \in \mathcal{T}_F$. We evaluate a HyperTSL formula φ with respect to a set of executions $E \subseteq \mathcal{C}^\omega \times \mathcal{I}^\omega$, an execution assignment function Π, and a point in time i. We omit the cases for $\neg$, $\wedge$, $\bigcirc$, and $\mathcal{U}$, which follow closely the respective cases in TSL.

$$\begin{array}{lll}
\Pi, E, i \models_{\langle\cdot\rangle} [\![\mathsf{c} \hookleftarrow \tau^f]\!]_\pi & \text{iff} & \#_1(\Pi(\pi))\ i\ \mathsf{c} \equiv \tau^f \\
\Pi, E, i \models_{\langle\cdot\rangle} (p\,\tau_1 \ldots \tau_n)_\pi & \text{iff} & \eta_{\langle\cdot\rangle}(\#_1(\Pi(\pi)), \#_2(\Pi(\pi)), i, p\,\tau_1 \ldots \tau_n) \\
\Pi, E, i \models_{\langle\cdot\rangle} \exists \pi.\, \varphi & \text{iff} & \exists e \in E.\, \Pi[\pi \mapsto e], E, i \models_{\langle\cdot\rangle} \varphi \\
\Pi, E, i \models_{\langle\cdot\rangle} \forall \pi.\, \varphi & \text{iff} & \forall e \in E.\, \Pi[\pi \mapsto e], E, i \models_{\langle\cdot\rangle} \varphi
\end{array}$$

Given an assignment function $\langle\cdot\rangle : \Sigma_F \to \mathcal{F}$, a set of executions E satisfies a HyperTSL formula φ, written $E \models_{\langle\cdot\rangle} \varphi$, if $\emptyset, E, 0 \models_{\langle\cdot\rangle} \varphi$.

8.1.2 HyperTSL$_{\text{rel}}$

In HyperTSL$_{\text{rel}}$, predicate terms are defined with respect to the set of executions variables V_π. If $\tau^1, \ldots, \tau^n$ are function terms, $\pi_1, \ldots \pi_n \in V_\pi$, and p is an n-ary predicate symbol, then $p\ \tau^1_{\pi_1}\ \ldots\ \tau^n_{\pi_n}$ is a predicate term. The syntax of HyperTSL$_{\text{rel}}$ is that of HyperTSL with the exception that we denote predicate terms as $\tau^p_{\pi_1,\ldots,\pi_n}$ instead of τ^p_π. The semantics of HyperTSL$_{\text{rel}}$ is that of HyperTSL except for the semantics of the predicate, which we define as follows.

$$\begin{array}{ll}
\Pi, E, i \models_{\langle\cdot\rangle} p\ \tau^1_{\pi_1}\ \ldots\ \tau^n_{\pi_n} \quad \text{iff} \quad \langle p\rangle\ v_1\ \ldots\ v_n \\
\text{where } v_j = \eta_{\langle\cdot\rangle}(\#_1(\Pi(\pi_j)), \#_2(\Pi(\pi_j)), i, \tau^j) \text{ for } 1 \le j \le n
\end{array}$$

Satisfaction of a HyperTSL$_{\text{rel}}$ formula is defined as for HyperTSL.

8.2 Specifying Hyperproperties in Smart Contracts

In this section, we discuss how HyperTSL and HyperTSL$_{\text{rel}}$ can express hyperproperties of smart contracts. As running example, we again use a simple voting protocol. Compared with the voting specification in →Section 7.1.1, this time, we focus on how to handle the vote, not that everybody only votes once. We assume that there are finitely many candidates. For this simple example, we restrict ourselves to the case of two candidates. →page 84

The specification of the contract is given in Specification 8.3. It describes an election where users can vote for either candidate `A` or candidate `B` by calling methods `voteA` or `voteB`, respectively. The contract can be closed by the owner by calling method `close`. The contract has two fields, `votesA` and `votesB`, to store the number of votes recorded for the candidates. Furthermore, `winner` holds the current winner chosen by the contract. The winner is updated in every step in which the contract

```
Methods: voteA, voteB, close, reveal
Fields: votesA, votesB
Functions: addOne
Predicates: >, =
Constants: owner(), A(), B()
Inputs: sender

--- Assumptions ---
□ ¬(votesA > votesB ∧ votesB > votesA);
¬(votesA > votesB) ∧ ¬(votesB > votesA);

--- Requirements ---
□(close → sender = owner());
□(voteA ∨ voteB → ⊟ ¬close);

--- Obligations ---
□(voteA → ⟦votesA ↤ votesA + 1⟧);
□(voteB → ⟦votesB ↤ votesB + 1⟧);
□(¬voteA → ⟦votesA ↤ votesA⟧);
□(¬voteB → ⟦votesB ↤ votesB⟧);

□((voteA ∨ voteB) → ⟦winner ↤ A()⟧ ∨ ⟦winner ↤ B()⟧);
□((voteA ∨ voteB) ∧ votesA > votesB → ⟦winner ↤ A()⟧);
□((voteA ∨ voteB) ∧ votesB > votesA → ⟦winner ↤ B()⟧);
□(⟠ close → ⟦winner ↤ winner⟧);
```

Specification 8.3: Specification of handling a vote in a voting contract.

receives a vote, so either $[\![\mathtt{winner} \leftarrowtail \mathtt{A()}]\!]$ or $[\![\mathtt{winner} \leftarrowtail \mathtt{B()}]\!]$ holds. `A()` and `B()` are constants. For a successful synthesis, the specification also includes arithmetic assumption on the $>$ predicate.

Not all of the properties desired of such a contract are expressible in TSL. Especially when it comes to fairness properties, we need to state hyperproperties. We show how hyperproperties in a voting contract can be expressed in HyperTSL and HyperTSL$_{\text{rel}}$. We use the following syntactic sugar.

$$\begin{aligned} \mathit{sameWinner}(\pi, \pi') \coloneqq {} & ([\![\mathtt{winner} \leftarrowtail \mathtt{A()}]\!]_{\pi} \leftrightarrow [\![\mathtt{winner} \leftarrowtail \mathtt{A()}]\!]_{\pi'}) \wedge \\ & ([\![\mathtt{winner} \leftarrowtail \mathtt{B()}]\!]_{\pi} \leftrightarrow [\![\mathtt{winner} \leftarrowtail \mathtt{B()}]\!]_{\pi'}) \end{aligned}$$

The formula states that two executions π and π' choose the same winner.

Determinism. As first property, we state that the winner of the election should be determined by the sequence of votes received.

$$\forall \pi, \pi'.\, \mathit{sameWinner}(\pi, \pi')\, \mathcal{W}\, \big((\mathtt{voteA}_{\pi} \nleftrightarrow \mathtt{voteA}_{\pi'}) \lor (\mathtt{voteB}_{\pi} \nleftrightarrow \mathtt{voteB}_{\pi'})\big)$$

The formula states that for any two executions, as long as they receive the exact same calls to voteA and voteB, they should always set the same winner. As an alternative definition of determinism, we could state a local version using the > predicate. We express that the choice of the winner is strictly determined by the current evaluation of the > predicate on the votes and the current vote.

$$\begin{aligned}\forall \pi, \pi'.\Box\big(((&\mathtt{votesA} > \mathtt{votesB})_{\pi} \leftrightarrow (\mathtt{votesA} > \mathtt{votesB})_{\pi'})\\ &\land ((\mathtt{votesB} > \mathtt{votesA})_{\pi} \leftrightarrow (\mathtt{votesB} > \mathtt{votesA})_{\pi'})\\ &\land (\mathtt{voteA}_{\pi} \leftrightarrow \mathtt{voteA}_{\pi'}) \land (\mathtt{voteB}_{\pi} \leftrightarrow \mathtt{voteB}_{\pi'})\\ &\rightarrow \mathit{sameWinner}(\pi, \pi')\big)\end{aligned} \tag{8.1}$$

This formula states in particular that if A and B received the same number of votes, the winner must be the same on both executions. Note that the formula implicitly entails that any other predicate/input does not influence the winner. For example, the evaluation of the predicate term `sender = owner()` should not influence the winner field. In HyperTSL$_{\text{rel}}$, we can express determinism by relating executions with predicates. Instead of abstracting from the concrete number of votes using the > predicate, we could state local determinism as follows.

$$\forall \pi, \pi'.\Box\big(\mathtt{votesA}_{\pi} = \mathtt{votesA}_{\pi'} \land \mathtt{votesB}_{\pi} = \mathtt{votesB}_{\pi'} \rightarrow \mathit{sameWinner}(\pi, \pi')\big)$$

Symmetry. A prominent fairness condition in systems with multiple agents is *symmetry*, which requires that agents are treated symmetrically. In the voting case, this means, e.g., that if two traces swap the votes for A and B, then the winner must also be swapped.

$$\begin{aligned}\forall \pi, \pi'.\, \big((&[\![\mathtt{winner} \hookleftarrow \mathtt{A()}]\!]_{\pi} \leftrightarrow [\![\mathtt{winner} \hookleftarrow \mathtt{B()}]\!]_{\pi'})\\ &\land ([\![\mathtt{winner} \hookleftarrow \mathtt{B()}]\!]_{\pi} \leftrightarrow [\![\mathtt{winner} \hookleftarrow \mathtt{A()}]\!]_{\pi'})\big)\\ &\mathcal{W}\, \big((\mathtt{voteA}_{\pi} \nleftrightarrow \mathtt{voteB}_{\pi'}) \lor (\mathtt{voteB}_{\pi} \nleftrightarrow \mathtt{voteA}_{\pi'})\big)\end{aligned} \tag{8.2}$$

The fairness property implies that the winner cannot be brute-forced to always be candidate A or candidate B in case of a tie.

No harm. A typical monotonicity criterion in elections states that a vote for the currently leading candidate should not harm the candidate. Translated to our context, this means that a vote for candidate A does not lead to B being the winner instead of A. We formulate this condition as follows in HyperTSL.

$$\begin{aligned}\forall \pi, \pi'.\big(\bigwedge_{x\in\mathcal{T}_P} x_\pi \leftrightarrow x_{\pi'}\big)\ \mathcal{U}\Big(\bigcirc\Box\big(\bigwedge_{x\in\mathcal{T}_P} x_\pi \leftrightarrow x_{\pi'}\big) \\ \wedge\ \mathtt{voteA}_\pi \wedge \mathtt{voteB}_{\pi'} \wedge \big(\bigwedge_{x\in(\mathcal{T}_P\setminus\{\mathtt{voteA},\mathtt{voteB}\})} x_\pi \leftrightarrow x_{\pi'}\big)\Big) \\ \rightarrow \Box(\llbracket \mathtt{winner} \leftarrowtail \mathtt{A()} \rrbracket_{\pi'} \rightarrow \llbracket \mathtt{winner} \leftarrowtail \mathtt{A()} \rrbracket_{\pi})\end{aligned} \tag{8.3}$$

The formula describes two executions on which all predicate terms evaluate to the same value except for one point in time at which the first execution receives a vote for A while the second receives a vote for B. Then, A is the winner in the first trace as least as often as A is in the second one.

Elections with Multiple Candidates. HyperTSL$_{\text{rel}}$ can specify hyperproperties in contracts with an unknown number of candidates. Assume that instead of voteA and voteB, the contract has a single method vote, which receives an argument cand indicating which candidate the caller is voting for. Now, the winner field ranges over a domain of unknown size. We express determinism as follows.

$$\forall \pi, \pi'.\Box\big((\mathtt{winner}_\pi = \mathtt{winner}_{\pi'})\ \mathcal{W}(\neg(\mathtt{arg@cand}_\pi = \mathtt{arg@cand}_{\pi'}))\big) \tag{8.4}$$

8.3 Synthesis from HyperTSL Specifications

In this section, we define the synthesis problem of HyperTSL and argue why HyperTSL is better suited for synthesis than HyperTSL$_{\text{rel}}$. HyperTSL inherits the undecidable realizability problem from TSL. We show that $\forall^*$ HyperTSL synthesis can be soundly approximated by HyperLTL synthesis, i.e, a strategy for the HyperLTL approximation can be translated to a strategy in HyperTSL. The $\exists^*$ fragment of HyperTSL synthesis can be approximated by LTL satisfiability to prove unrealizability.

Definition 8.1. A HyperTSL formula φ is *realizable* iff there exists a strategy $\sigma : (2^{\mathcal{T}_P})^+ \rightarrow C$ such that for every interpretation $\langle\cdot\rangle : \Sigma_P \rightarrow \mathcal{F}$, the set constructed from all input streams satisfies φ, i.e., $\{(\sigma(\iota), \iota) \mid \iota \in \mathcal{I}^\omega\} \models_{\langle\cdot\rangle} \varphi$.

The above definition defines the realizability problem of HyperTSL by generalizing the definition of →TSL realizability, similarly to how HyperLTL realizability generalizes LTL realizability. The strategy has the same type as a TSL strategy, i.e., it generates an execution by reacting to the predicate evaluations for the current input stream. The resulting set of executions must satisfy the HyperTSL formula.

→page 79

In HyperTSL$_{\text{rel}}$, formulas contain predicates that relate multiple executions. However, a system can still only react to the inputs it receives, i.e., it cannot include the evaluation of such predicates in its decision making. A definition of the realizability problem could therefore only contain predicates not ranging over multiple executions. This makes many formulas unrealizable or only realizable by trivial strategies most likely not intended by the developer.

Consider again Formula (8.4) in Section 8.2. The evaluation of the terms $\texttt{winner}_\pi = \texttt{winner}_{\pi'}$ and $\texttt{arg@cand}_\pi = \texttt{arg@cand}_{\pi'}$ depends on the chosen executions π and π' and can therefore not be known to the system. The predicate symbol $=$ is uninterpreted and could be implemented with any binary predicate. A strategy must be winning for all interpretations. The only winning strategy is therefore the trivial one to always copy the value of `arg@cand` to the `winner` field. That way, the $=$ predicate must always evaluate the same on the two predicate terms.

The above observations make HyperTSL the better candidate to extend TSL synthesis to hyperproperties. This is only the case for synthesis, though, and as long as predicates and functions remain uninterpreted. If we considered HyperTSL$_{\text{rel}}$ synthesis with the theory of equality, much more meaningful strategies would be possible. HyperTSL$_{\text{rel}}$ might also be useful for model checking, where formulas can contain predicates that are not part of the system. So far, there are only few preliminary works that study TSL synthesis with theories [171, 90]. TSL model checking has not been studied at all.

8.3.1 Realizability of $\forall^*$ HyperTSL

We show that the fact that TSL synthesis can by approximated by LTL synthesis carries over to $\forall^*$ HyperTSL. Given a TSL formula ψ, let $[\psi]_{\text{atomic}}$ be the LTL formula obtained by the →LTL approximation of ψ. Given an execution $e = (\varsigma, \iota) \in \mathcal{C}^\omega \times \mathcal{I}^\omega$ and an interpretation $\langle\cdot\rangle$, we define the corresponding LTL trace $[e]^{\langle\cdot\rangle}_{\text{atomic}}$ over AP_ψ. →page 80

$$[e]^{\langle\cdot\rangle}_{\text{atomic}}(i) := \{a_{\tau^p} \mid \eta_{\langle\cdot\rangle}(\varsigma, \iota, i, \tau^p)\} \cup \{a_{\mathsf{x}\_to\_\tau^f} \mid \varsigma\ i\ \mathsf{x} \equiv \tau^f\}$$

Theorem 1 in [93] states that for a TSL formula ψ, if $[\psi]_{\text{atomic}}$ is realizable, then ψ is realizable. Actually, their proof shows the following stronger result.

Proposition 8.1 (Proof of Theorem 1 in [93]). *For any TSL formula ψ, execution e, interpretation $\langle\cdot\rangle$, and point in time i, it holds $e, i \models_{\langle\cdot\rangle} \psi$ iff $[e]^{\langle\cdot\rangle}_{\text{atomic}}, i \models [\psi]_{\text{atomic}}$.*

This results entails that realizability of $[\psi]_{\text{atomic}}$ implies realizability of ψ; the opposite direction does not hold (see →Section 6.2). Every execution e can be mapped to a trace $[e]^{\langle\cdot\rangle}_{\text{atomic}}$, but there are traces t over AP_ψ such that there is no execution e with $[e]^{\langle\cdot\rangle}_{\text{atomic}} = t$. The approximation is still valuable, because strategies found for the LTL →page 80

approximations carry over to strategies for the TSL specification. We show that this result can be lifted to $\forall^*$ HyperTSL and $\forall^*$ HyperLTL. Given a set of executions E, we set $[E]^{\langle\cdot\rangle}_{\text{atomic}} = \{[e]^{\langle\cdot\rangle}_{\text{atomic}} \mid e \in E\}$ and also lift $[\cdot]_{\text{atomic}}$ to HyperTSL formulas by setting $[\tau_\pi]_{\text{atomic}} = ([\tau]_{\text{atomic}})_\pi$ with τ being either a predicate term or an update term.

Lemma 8.2. *For any HyperTSL formula φ, set of executions E, and interpretation $\langle\cdot\rangle$, it holds $E \models_{\langle\cdot\rangle} \varphi$ iff $[E]^{\langle\cdot\rangle}_{atomic} \models [\varphi]_{atomic}$.*

Proof. Let $\varphi = (\exists^*\forall^*)^k.\,\psi$ be a HyperTSL formula consisting of k blocks of $\exists^*\forall^*$ quantifiers. For the first direction, let $E \models_{\langle\cdot\rangle} \varphi$. We show $[E]^{\langle\cdot\rangle}_{\text{atomic}} \models [\varphi]_{\text{atomic}}$. Let $\varphi[i]$ be the subformula of φ starting from the ith quantifier block with $1 \leq i \leq k$. We keep the invariant to only construct trace assignments Π such that there exists an execution assignment $\hat{\Pi}$ with $\Pi(\pi) = [\hat{\Pi}(\pi)]^{\langle\cdot\rangle}_{\text{atomic}}$ and $\hat{\Pi}, E, 0 \models_{\langle\cdot\rangle} \varphi[i]$. When choosing the witness traces for the existential variables $\pi^{i+1}_1, \ldots \pi^{i+1}_n$ of the $(i+1)$th quantifier block, we choose $\Pi(\pi^{i+1}_j) = [e^{i+1}_j]^{\langle\cdot\rangle}_{\text{atomic}}$ where e^{i+1}_j is assigned to π^{i+1}_j by the proof of satisfaction of $\varphi[i]$ with respect to the current execution assignment $\hat{\Pi}$. Now, by Proposition 8.1 and a simple induction over the structure of ψ, we get $\Pi, [E]^{\langle\cdot\rangle}_{\text{atomic}}, 0 \models \psi$. The argument is similar for the other direction since for every $t \in [E]^{\langle\cdot\rangle}_{\text{atomic}}$, there exists an $e \in E$ such that $[e]^{\langle\cdot\rangle}_{\text{atomic}} = t$. □

Note that the above lemma does not extend to HyperTSL$_{\text{rel}}$ as predicates over multiple executions cannot be mapped to HyperLTL. The lemma entails the following theorem. We will also reuse the lemma in Section 8.4.

Theorem 8.3. *Let φ be a $\forall^*$ HyperTSL formula. If $[\varphi]_{atomic}$ is realizable, then φ is realizable.*

Proof. Let $AP_\varphi = AP_{in} \cup AP_{out}$ be the atomic propositions obtained in the translation $[\varphi]_{\text{atomic}}$, where AP_{in} are the propositions generated for predicate terms and AP_{out} the propositions for update terms, respectively. Let $\sigma_{\text{HyperLTL}} : (2^{AP_{in}})^+ \rightarrow 2^{AP_{out}}$ be the realizing strategy for $[\varphi]_{\text{atomic}}$. We claim that the following strategy σ realizes φ.

$$\sigma\ (P_1 \ldots P_i)\ \mathsf{c} := \tau^f \quad \text{iff} \quad [[\![\mathsf{c} \leftarrowtail \tau^f]\!]]_{\text{atomic}} \in \sigma_{\text{HyperLTL}}([P_1]_{\text{atomic}} \ldots [P_i]_{\text{atomic}})$$

Here, we lift $[\cdot]_{\text{atomic}}$ to sets by $[P_j]_{\text{atomic}} = \{[\tau^p]_{\text{atomic}} \mid \tau^p \in P_j\}$. By definition of →*cellProps*, the function term τ^f is unique for every i and every cell c. We show that σ realizes φ. Let $\langle\cdot\rangle$ be any interpretation. Let $E = \{(\sigma(\iota), \iota) \mid \iota \in \mathcal{I}^\omega\}$ be the set of executions obtained from σ and $T = traces(\sigma)$ be the set of traces obtained from σ_{HyperLTL}. Notice that $[E]^{\langle\cdot\rangle}_{\text{atomic}} \subseteq T$, but T may not be a subset of $[E]^{\langle\cdot\rangle}_{\text{atomic}}$. Since $T \models [\varphi]_{\text{atomic}}$ and since universal properties are downwards-closed, also $[E]^{\langle\cdot\rangle}_{\text{atomic}} \models [\varphi]_{\text{atomic}}$. Therefore, by Lemma 8.2, we have $E \models_{\langle\cdot\rangle} \varphi$. □

→page 80

This theorem approximates $\forall^*$ HyperTSL synthesis by $\forall^*$ HyperLTL synthesis. Even for HyperLTL, this problem is undecidable [86]. However, there exists an implementation of bounded synthesis for $\forall^*$ HyperLTL in the tool BoSyHyper [86]. This approach searches for smallest systems implementing the formula. The efficient reduction from $\forall^*$ HyperTSL to HyperLTL opens the door to apply BoSyHyper to HyperTSL.

8.3.2 Realizability of $\exists^*$ HyperTSL

Theorem 8.3 uses the downwards closedness of $\forall^*$ HyperLTL properties. This is necessary since a strategy for HyperTSL produces "fewer" traces than a strategy for the translation of the formula to HyperLTL does. Consequently, the proof does not extend to HyperTSL formulas with existential quantifiers. We show how to translate the synthesis problem of $\exists^*$ HyperTSL formulas to a TSL satisfiability problem, which we can approximate by LTL satisfiability checking using the translation from TSL to LTL from [93]. In contrast to the case of $\forall^*$ HyperTSL formulas, the approximation of existential properties in LTL can be used to show unrealizability instead of realizability.

Theorem 8.4. *Given an $\exists^*$ HyperTSL formula φ, there exists a TSL formula ψ_{TSL} such that φ is realizable iff ψ is satisfiable.*

Proof. Let $\varphi = \exists \pi_1, \ldots, \pi_n.\, \psi$. We encode the fact that all traces in a model of φ are producible by a strategy:

$$\varphi_{\text{strat}} := \forall \pi, \pi'. \left(\bigwedge_{\tau^u \in \mathcal{T}_U} \tau^u_\pi \leftrightarrow \tau^u_{\pi'} \right) \mathcal{W} \left(\bigvee_{\tau^p \in \mathcal{T}_P} \tau^p_\pi \not\leftrightarrow \tau^p_{\pi'} \right)$$

The formula states that while two executions have the same predicate evaluations, they have to perform the same updates. Formula $\varphi' = \exists \pi_1, \ldots, \pi_n.\, \psi \wedge \varphi_{\text{strat}}$ ensures that the executions chosen as witnesses for $\pi_1, \ldots, \pi_n$ can be arranged in a strategy tree. Thus, φ and φ' are equi-realizable. The additional conjunct also ensures that φ' is satisfiable iff it is realizable. Formula φ' is satisfiable iff it is satisfiable by n executions. The $\forall$ quantifiers in φ' quantify over these n executions. We can therefore create an equisatisfiable $\exists^n$ formula by "unrolling" the $\forall$ quantifiers. Let $\varphi_{\text{strat}} = \forall \pi, \pi'.\, \psi_{\text{strat}}$. We define:

$$\varphi_{\text{new}} := \exists \pi_1, \ldots, \pi_n.\, \psi \wedge \bigwedge_{1 \le i,j \le n} \psi_{\text{strat}}[\pi \mapsto \pi_i, \pi' \mapsto \pi_j]$$

We use the notation $\psi[\pi \mapsto \pi']$ to indicate ψ with every occurrence of π replaced by π'. Now, we have that φ is realizable iff φ_{new} is satisfiable. The above formula is satisfiable iff the TSL formula ψ_{TSL} is satisfiable, where we construct ψ_{TSL} by creating n copies of each input and each cell. $\square$

It is an open question whether the two approximations for $\forall^*$ HyperTSL and $\exists^*$ HyperTSL can be combined to approximate formulas with quantifier alternations. The challenge here is that universal properties are best over-approximated to obtain realizability results, whereas existential properties would need an under-approximation.

8.4 Pseudo Hyperproperties

We have shown that universal HyperTSL formulas can be approximated by HyperLTL synthesis. This result is, as of now, mostly of theoretic interest. The realizability problem of $\forall^*$ HyperLTL is undecidable [86]. Bounded hyperproperty synthesis is also still in its infancy.

In this and the following section, we present a more feasible approach to synthesize smart contract control flows. We use the fact that we do not synthesize a system from hyperproperties only, but in combination with trace properties describing the functional properties of the contract. We therefore extend the synthesis approach described in →Section 7.1 with methods for HyperTSL. In this section, we discuss the challenges of combining the synthesis from pastTSL with HyperTSL specifications. We then analyze whether the combination of hyperproperties with trace properties leads to "pseudo" hyperproperties. In Section 8.5, we synthesize the most general system from the trace properties and resolve free choices with a repair-like algorithm. →page 84

8.4.1 Combining PastTSL Synthesis with HyperTSL

If we inspect the winning region of the voting specification from Specification 8.3, we notice that the system has free choices. The state machine is depicted in Figure 8.1. The TSL specification leaves the free choice which candidate is the winner when neither one has the majority of votes. Strategies permitted by the winning region are, for example, to always choose A as winner, or go for the candidate who got the last vote.

Which strategies are considered to be suitable is described by the HyperTSL formulations of local determinism (Formula (8.1) in Section 8.2), symmetry (Formula (8.2)), and the no harm property (Formula (8.3)). Our goal is to combine the TSL specification with the HyperTSL specifications to obtain a satisfactory strategy. Since the specification of the voting contract abstracts from the concrete number of votes with the $>$ predicate, we need to add the following assumption to make the combination of the specifications realizable.

$$\begin{aligned}\forall \pi, \pi'.\ \big(((\mathtt{votesA} > \mathtt{votesB})_\pi \leftrightarrow (\mathtt{votesA} > \mathtt{votesB})_{\pi'}) \\ \land ((\mathtt{votesB} > \mathtt{votesA})_\pi \leftrightarrow (\mathtt{votesB} > \mathtt{votesA})_{\pi'})\big) \\ \mathcal{W}\,(\mathtt{voteA}_\pi \not\leftrightarrow \mathtt{voteA}_{\pi'} \lor \mathtt{voteB}_\pi \not\leftrightarrow \mathtt{voteB}_{\pi'})\end{aligned}$$

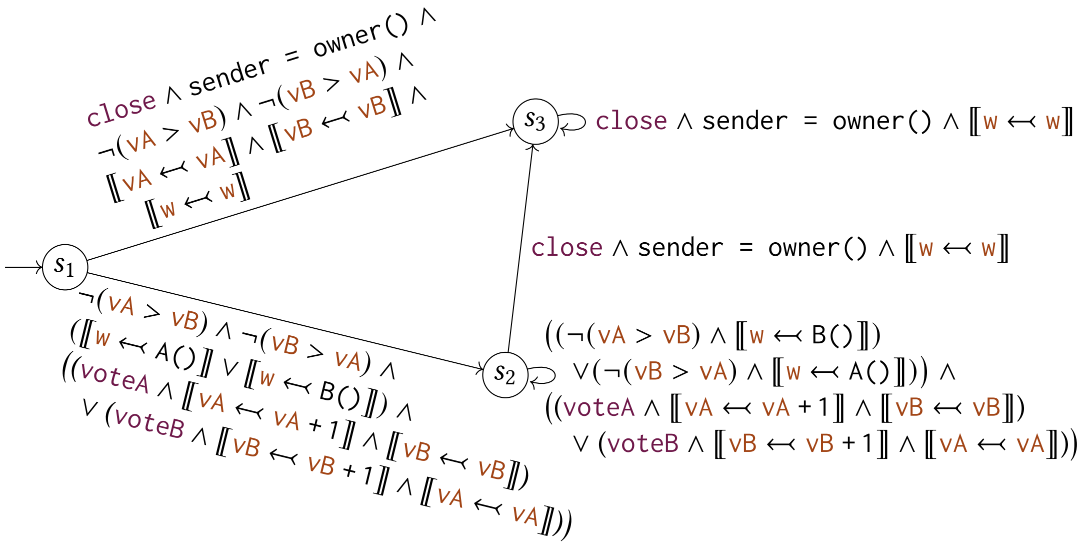

close ∧ sender = owner() ∧
¬(vA > vB) ∧ ¬(vB > vA) ∧
⟦vA ↢ vA⟧ ∧ ⟦vB ↢ vB⟧ ∧
⟦w ↢ w⟧
s_3
close ∧ sender = owner() ∧ ⟦w ↢ w⟧
close ∧ sender = owner() ∧ ⟦w ↢ w⟧
s_1
¬(vA > vB) ∧ ¬(vB > vA) ∧
(⟦w ↢ A()⟧ ∨ ⟦w ↢ B()⟧) ∧
((voteA ∧ ⟦vA ↢ vA + 1⟧ ∧ ⟦vB ↢ vB⟧)
∨ (voteB ∧ ⟦vB ↢ vB + 1⟧ ∧ ⟦vA ↢ vA⟧))
s_2
((¬(vA > vB) ∧ ⟦w ↢ B()⟧)
∨(¬(vB > vA) ∧ ⟦w ↢ A()⟧)) ∧
((voteA ∧ ⟦vA ↢ vA + 1⟧ ∧ ⟦vB ↢ vB⟧)
∨ (voteB ∧ ⟦vB ↢ vB + 1⟧ ∧ ⟦vA ↢ vA⟧))

Figure 8.1: Winning region for Specification 8.3 as synthesized by SCSynt [91]. We write w instead of winner, and vA and vB instead of votesA and votesB, respectively.

The property states that the evaluation of the > predicate is the same for every two executions as long as they receive the same sequence of votes. The property is satisfied when implementing +1 as increment and > as "greater than".

When combining the TSL specification of the voting contract with local determinism, symmetry, and the no harm property, we observe that there is only one valid strategy left to resolve a tie. Local determinism states that the winner may only depend on the greater predicate on the votes and the vote cast in the current step. This forbids to take into account the past of the trace, e.g., by choosing the winner differently in the first and in the second step. Symmetry forbids to hard code and always let the same candidate win in case of a tie. Together with local determinism, the winner must therefore truly depend on the current vote. In combination with local determinism and symmetry, the no harm property forbids to let A win if the current vote is for B (and vice versa). This leaves as only option to resolve a tie to let the candidate win who got the current vote. In this case, we could therefore replace the hyperproperties with a trace property describing exactly this strategy, which leads to a much easier synthesis problem.

8.4.2 Definition of Pseudo Hyperproperties

As a human, it can be hard ro recognize when a hyperproperty together with a trace property describes "only" a trace property. We therefore propose to preprocess the specification by checking whether the hyperproperties in conjunction with the trace properties effectively define a trace property. To do so, we introduce the notion of pseudo hyperproperties. We first investigate the problem for the general definition based on equivalence checking, and subsequently consider the special case of synthesis.

Definition 8.2. A hyperproperty H is a pseudo hyperproperty iff there is a trace property P such that

$$\forall T \subseteq (2^{AP})^{\omega}.\, T \in H \text{ iff } \forall t \in T.\, t \in P$$

If H is a pseudo hyperproperty, H describes the trace property $P = \bigcup_{T \in H} T$. For proofs we use the following fact.

Proposition 8.5. *H is a pseudo hyperproperty iff H is closed under union and subsets, i.e., if $T, T' \in H$, then $T \cup T' \in H$; and if $T \in H$ and $T' \subseteq T$, then $T' \in H$.*

The downwards closure property in the above proposition implies that only hyperproperties expressible with $\forall^*$ formulas can be pseudo hyperproperties. The following proposition establishes the convenient fact that if a $\forall^*$ HyperTSL formula φ describes a pseudo hyperproperty, then the formula obtained by using only a single execution variable in the body of the formula is equivalent to φ. The proposition is close to the corresponding result for HyperLTL, which has been observed in context of using HyperLTL synthesis for the synthesis of linear distributed architectures [86]. Given a HyperTSL formula $\varphi = \forall \pi_1, \ldots, \pi_n.\, \psi$, we define its $\forall^1$ counterpart as $\varphi[\pi] \coloneqq \forall \pi.\, \psi[\pi_1 \mapsto \pi, \ldots, \pi_n \mapsto \pi]$.

Proposition 8.6. *A $\forall^*$ HyperTSL formula φ describes a pseudo hyperproperty iff $\varphi \equiv \varphi[\pi]$.*

Proof. Following from Proposition 8.5, we have that a hyperproperty H is a pseudo hyperproperty iff

$$\forall T \subseteq (2^{AP})^{\omega}.\, T \in H \text{ iff } \forall t \in T.\, \{t\} \in H$$

Thus, for any set of executions E and any interpretation $\langle\cdot\rangle$, we have $E \models_{\langle\cdot\rangle} \varphi$ iff $\forall e \in E.\, \{e\} \models_{\langle\cdot\rangle} \varphi$ iff $E \models_{\langle\cdot\rangle} \varphi[\pi]$. □

The proposition describes how to check if a HyperTSL property is a pseudo hyperproperty, namely by checking if φ is equivalent to $\varphi[\pi]$. If it is, the TSL formula $\varphi[\pi]$ can be used for synthesis. The check is undecidable for HyperTSL, however.

Theorem 8.7. *It is undecidable whether a HyperTSL formula φ describes a pseudo hyperproperty.*

Proof. The proof follows from undecidability of the satisfiability problem of TSL [90]. The formula $\varphi = \forall \pi, \pi'.(p\ a)_{\pi} \leftrightarrow (p\ a)_{\pi'}$ is not a pseudo hyperproperty as there exists an interpretation of p and sets E and E' such that E consists of executions for which $p\ a$

always evaluates to *true*, and E' consists of executions for which $p\ a$ always evaluates to *false*. E and E' are not closed under union for φ. Now, let a TSL formula ψ be given for which we assume w.l.o.g. that p is a fresh predicate and a is a fresh input. Since $\forall \pi, \pi'.\, \mathit{false}$ is a pseudo hyperproperty, the formula $\forall \pi, \pi'.\, ((p\ a)_\pi \leftrightarrow (p\ a)_{\pi'}) \land \psi_\pi$ is a pseudo hyperproperty iff ψ is unsatisfiable. Here, ψ_π denotes the formula obtained by lifting ψ to a HyperTSL formula by annotating predicate and update terms with the trace variable π. □

Fortunately, the above negative result comes with the remedy that we can approximate the problem in HyperLTL, which is a result of Lemma 8.2.

Theorem 8.8. *Let φ be a HyperTSL formula. If $[\varphi]_{atomic}$ is a pseudo hyperproperty, then so is φ.*

Proof. Using Proposition 8.5, we prove that the set of sets of traces that are accepted by φ is closed under union and subsets. This follows from Lemma 8.2 together with the observation that for every set of executions E and interpretation $\langle\cdot\rangle$, $[E]^{\langle\cdot\rangle}_{\text{atomic}} = \bigcup_{e \in E} [e]^{\langle\cdot\rangle}_{\text{atomic}}$, i.e., $[E \cup E']^{\langle\cdot\rangle}_{\text{atomic}} = [E]^{\langle\cdot\rangle}_{\text{atomic}} \cup [E']^{\langle\cdot\rangle}_{\text{atomic}}$, and $E \subseteq E'$ iff $[E]^{\langle\cdot\rangle}_{\text{atomic}} \subseteq [E']^{\langle\cdot\rangle}_{\text{atomic}}$. □

Using the above result, we can translate a $\forall^*$ HyperTSL formula to HyperLTL and check its equivalence with the corresponding $\forall^1$ formula. For HyperLTL, this check is decidable, because of the decidability of satisfiability for the $\exists^*\forall^*$ fragment of HyperLTL [86]. A HyperLTL formula $\varphi = \forall \pi_1, \ldots, \pi_n.\, \psi$ is equivalent to its $\forall^1$ counterpart $\varphi[\pi]$, iff $\varphi[\pi] \rightarrow \varphi$ (because of the semantics of the $\forall$). By merging the quantifiers, this implication is valid iff the formula

$$\exists \pi_1, \ldots, \pi_n.\, \forall \pi.\, \neg\psi \land \psi[\pi_1 \mapsto \pi, \ldots \pi_n \mapsto \pi]$$

is unsatisfiable. Thus, if the check returns UNSAT, the HyperTSL formula is guaranteed to describe a pseudo hyperproperty, so we can replace the hyperproperty with its TSL version.

8.4.3 Pseudo Hyperproperties in Synthesis

So far, we described the check for pseudo hyperproperties in terms of satisfiability checking. This can be useful to detect superfluous specifications or mistakes. For example, the developer could have already specified in TSL that in case of a tie, the current vote determines the winner.

$$\Box(\neg(\texttt{votesA} > \texttt{votesB}) \land \neg(\texttt{votesB} > \texttt{votesA}) \rightarrow (⟦\texttt{winner} \leftarrowtail \texttt{A()}⟧ \leftrightarrow \texttt{voteA}))$$

With this specification, the hyperproperties stated in Section 8.2 are entailed by the specification. In the case of the no harm property, for example, the check with EAHyper reveals correctly within 0.003 seconds that the conjunction of the trace properties with the no harm property is a pseudo hyperproperty.

Without the above TSL specification, however, the check for the conjunction of local determinism, symmetry, and the no harm property is labeled as a hyperproperty by EAHyper, even though there is only one possible strategy. The reason for that is the difference between realizability and satisfiability. Consider the following two traces.

$$\begin{aligned}&(1)\ \{\texttt{voteA}, [\![\texttt{winner} \leftarrowtail \texttt{A()}]\!]\}^\omega\\ &(2)\ \{\texttt{voteB}, [\![\texttt{winner} \leftarrowtail \texttt{A()}]\!]\}^\omega\end{aligned}$$

In the first trace, neither `votesA` > `votesB` nor `votesB` > `votesA` holds, and there is always a vote for `A`, who is always the winner. The second trace is similar but there is always a vote for `B`. As single sets, both traces satisfy all three hyperproperties. Together, they do not (because of the symmetry requirement). Therefore, EAHyper returns that the conjunction of the properties is not a pseudo hyperproperty (even in combination with the full contract specification). When solving the realizability problem, however, we do not need to find an *equivalent* trace property, it is enough to find a trace property which evaluates the same on all sets generated by strategies. We therefore adapt the notion of pseudo hyperproperties to realizability. We give the definition for hyperproperties of traces over $AP_{in} \cup AP_{out}$, but the definition applies to traces of any type, particularly also to hyperproperties over TSL-like executions.

Definition 8.3. Let H be a hyperproperty over atomic propositions $AP = AP_{in} \cup AP_{out}$. H is a pseudo hyperproperty in the context of realizability if there is a trace property P such that

$$\forall \sigma : (2^{AP_{in}})^+ \to 2^{AP_{out}}.\ traces(\sigma) \in H \text{ iff } \forall t \in traces(\sigma).\ t \in P$$

The definition implies that a HyperTSL formula φ is a pseudo hyperproperty in the context of realizability iff there is a TSL formula ψ such that a strategy σ realizes φ iff σ realizes ψ. Unfortunately, ψ may not be realized by the same strategies as $\varphi[\pi]$. To show why, we give an example in HyperLTL, but the reasoning carries over to HyperTSL. Consider the example of the following HyperLTL formulas over input i and output o, which are abstract, simplified versions of local determinism, symmetry, and the no harm property.

$$\begin{aligned}&\forall \pi, \pi'.\ (i_\pi \leftrightarrow i_{\pi'}) \to (o_\pi \leftrightarrow o_{\pi'})\\ &\forall \pi, \pi'.\ (i_\pi \not\leftrightarrow i_{\pi'}) \to (o_\pi \not\leftrightarrow o_{\pi'})\\ &\forall \pi, \pi'.\ i_\pi \land \neg i_{\pi'} \land o_{\pi'} \to o_\pi\end{aligned}$$

The properties are not temporal, so they are realizable iff there are values o_1, o_2 such that $\{\{i, o_1\}^\omega, \{\neg i, o_2\}^\omega\}$ satisfies the three formulas. Let φ be their conjunction. There are only four possible assignments of o_1, o_2 to Boolean values, each of which corresponds to a possible strategy. Indeed, for every possible strategy, φ is satisfied iff the trace property $i \leftrightarrow o$ is satisfied. The same does not hold for the trace property *true*, which is equivalent to $\varphi[\pi]$. Unfortunately, the general problem is undecidable already for HyperLTL, due to the undecidability of the realizability problem of the $\forall^*$ fragment of HyperLTL.

Theorem 8.9. *Given a HyperLTL formula φ, it is in general undecidable if there exists an LTL formula ψ such that for all strategies σ, $traces(\sigma) \models \varphi$ iff $\forall t \in traces(\sigma).\, t \models \psi$.*

Proof. We show undecidability by a reduction from the $\forall^*$ HyperLTL realizability problem, which is undecidable [86]. Let ρ be a $\forall^*$ HyperLTL formula over $AP = AP_{in} \cup AP_{out}$. We assume w.l.o.g. that AP_{in} is non-empty. We define φ as follows, where o is an output proposition that does not occur in ρ.

$$\varphi := \rho \wedge \forall \pi, \pi'.\, o_\pi \leftrightarrow o_{\pi'}$$

We claim that ρ is unrealizable iff there exists an LTL formula ψ such that for all strategies σ, $traces(\sigma) \models \varphi$ iff $\forall t \in traces(\sigma).\, t \models \psi$. If ρ is unrealizable, then we choose $\psi := false$. Since ρ is unrealizable, it holds for all σ that $traces(\sigma) \not\models \varphi$ iff $\forall t \in traces(\sigma).\, t \models \psi$. For the other direction, assume that ρ is realizable and that there exists a suitable LTL formula ψ. Let σ be the realizing strategy of ρ. Since o is fresh for ρ, we can extend σ to σ_1 and σ_2, where σ_1 adds o to the first output (for any input), and σ_2 does not add o (also for any input). Both strategies realize φ, therefore, by our assumption, both strategies realize ψ. Now, let $i \in AP_{in}$ be any input. We define σ_3 as the strategy that adds o to the first output exactly if i holds in the first position of the input sequence. Since every trace generated by σ_3 is either a trace of σ_1 or σ_2, σ_3 realizes ψ. However, it does not realize φ, which contradicts our assumption. □

Lastly, we observe that we can decide if a HyperTSL formula is a pseudo hyperproperty if the formula contains as conjunct a local determinism formula like the one in our running example. We define local determinism in the general case as follows.

$$localDeterminism := \forall \pi, \pi'. \square\Big(\bigwedge_{\tau^p \in \mathcal{T}_P} (\tau^p_\pi \leftrightarrow \tau^p_{\pi'}) \rightarrow \bigwedge_{\tau^c \in \mathcal{T}_U} (\tau^c_\pi \leftrightarrow \tau^c_{\pi'})\Big)$$

Proposition 8.10. *For every $\forall^*$ HyperTSL formula φ over predicate terms $\mathcal{T}_P$ and update terms $\mathcal{T}_U$, if $\varphi \equiv localDeterminism \wedge \varphi'$, then deciding whether φ is a pseudo hyperproperty in the context of realizability is equivalent to a Boolean SAT problem.*

The above proposition follows from the observation that $\mathit{localDeterminism} \land \varphi'$ is realizable iff there is a positional strategy that always assigns the same cell updates for the same predicate evaluations, independently of the trace's past. In this case, there are only finitely many combinations of predicate evaluations and the problem becomes finite.

8.5 Resolving Choices with Repair

We discuss how the combination of a TSL specification with HyperTSL formulas can be synthesized. If the hyperproperty is not a pseudo hyperproperty, we propose to synthesize the winning region of the trace property and then check if free choices can be resolved such that the hyperproperty is satisfied. This is a repair algorithm additionally respecting the distinction between inputs and outputs. We first give a formal definition of the problem. As observed several times, the formal problem is undecidable for HyperTSL, but a sound approximation can be achieved through HyperLTL. We then discuss how to simplify the problem for $\forall^*$ formulas and present a prototype implementation, which successfully repairs the synthesized voting contract with respect to determinism, symmetry, and the no harm property. We first define what constitutes a free choices and refinements in →Mealy machines. →page 76

Definition 8.4. A *free choice* in a Mealy machine $(S, s_0, \Sigma_{in}, \Sigma_{out}, \delta)$ consists of a state $s \in S$ and an input $i \in \Sigma_{in}$ such that there are at least two outputs $o_1, o_2 \in \Sigma_{out}$ with $(s, i, o_1, s_1) \in \delta$ and $(s, i, o_2, s_2) \in \delta$ for some $s_1, s_2 \in S$.

Definition 8.5. A Mealy machine $\hat{\mathcal{M}}$ is a *refinement* of a Mealy machine $\mathcal{M}$ if $\hat{S} = S$, $\hat{s_0} = s_0$, $\hat{\Sigma}_{in} = \Sigma_{in}$, $\hat{\Sigma}_{out} = \Sigma_{out}$, and $\hat{\delta} \subseteq \delta$ such that for every $s \in S, i \in \Sigma_{in}$, if there are $o \in \Sigma_{out}$ and $s' \in S$ such that $\delta(s, i, o, s')$, then there are $\hat{o} \in \Sigma_{out}$ and $\hat{s}' \in \hat{S}$ such that $\hat{\delta}(s, i, \hat{o}, \hat{s}')$ and $\delta(s, i, \hat{o}, \hat{s}')$.

The above definition ensures that every strategy of $\hat{\mathcal{M}}$ is one of $\mathcal{M}$, and if $\mathcal{M}$ defines a set of strategies for a TSL property ψ, then a $\hat{\mathcal{M}}$ still describes at least one strategy for ψ. Our goal is to refine $\mathcal{M}$ such that $\hat{\mathcal{M}}$ models a HyperTSL property φ.

The system $\mathcal{M}$ is produced from the LTL approximation, it might therefore contain spurious traces which, for any interpretation, cannot be produced by a combination of an input stream and a computation. We define when such a system satisfies a HyperTSL formula.

Definition 8.6. Let φ be a HyperTSL formula over $\mathcal{T}_P, \mathcal{T}_U$ and let $\mathcal{M}$ be a Mealy machine over $\Sigma_{in} = 2^{[\mathcal{T}_P]_{\text{atomic}}}$, $\Sigma_{out} = 2^{[\mathcal{T}_U]_{\text{atomic}}}$. $\mathcal{M}$ *models* φ if for every interpretation $\langle\cdot\rangle$:

$$\{(\varsigma, \iota) \mid \exists t \in \mathit{traces}(\mathcal{M}).\, t = [(\varsigma, \iota)]^{\langle\rangle}_{\text{atomic}}\} \models_{\langle\rangle} \varphi$$

The definition creates, for each interpretation, the set of executions which have a trace through $\mathcal{M}$ and then checks if φ is satisfied on these sets. The definition thus ignores spurious traces that might be contained in $\mathcal{M}$. We show that we cannot check the satisfaction of a HyperTSL formula on such a system directly.

Lemma 8.11. *Given a HyperTSL formula φ, it is undecidable whether a Mealy machine $\mathcal{M}$ over $\Sigma_{in} = 2^{[\mathcal{T}_P]_{atomic}}$, $\Sigma_{out} = 2^{[\mathcal{T}_U]_{atomic}}$ models φ.*

Proof. We show that the problem is already undecidable for TSL formulas expressed in HyperTSL as $\forall\pi.\, \psi_\pi$, where ψ_π is the π-indexed version of a TSL formula ψ. We proceed by reduction from the unsatisfiability problem of TSL [90]. Let ψ be a TSL formula. ψ is unsatisfiable if for every $\langle\cdot\rangle$ and every ς, ι, it holds that $\varsigma, \iota \models_{\langle\cdot\rangle} \neg\psi$. Let $\mathcal{M}$ be the system that produces every trace over $2^{[\mathcal{T}_P]_{\text{atomic}}} \cup 2^{[\mathcal{T}_U]_{\text{atomic}}}$. Then, $\mathcal{M}$ models $\forall\pi.\, \neg\psi_\pi$ iff ψ is unsatisfiable. $\square$

However, owing to Lemma 8.2, for every $\forall^*$ HyperTSL formula φ, if a Mealy machine $\mathcal{M}$ models $[\varphi]_{\text{atomic}}$, then it also models φ.

Theorem 8.12. *Let a $\forall^*$ HyperTSL formula and a Mealy machine $\mathcal{M}$ over $\Sigma_{in} = 2^{[\mathcal{T}_P]_{atomic}}$, $\Sigma_{out} = 2^{[\mathcal{T}_U]_{atomic}}$ be given. If $\mathcal{M}$ models $[\varphi]_{atomic}$, then $\mathcal{M}$ models φ.*

Proof. For every $\langle\cdot\rangle$, let $\mathcal{M}_{\langle\cdot\rangle} = \{(\varsigma, \iota) \mid \exists t \in \mathit{traces}(\mathcal{M}).\, t = [(\varsigma, \iota)]^{\langle\cdot\rangle}_{\text{atomic}}\}$. By construction, $[\mathcal{M}_{\langle\cdot\rangle}]^{\langle\cdot\rangle}_{\text{atomic}} \subseteq \mathit{traces}(\mathcal{M})$. Since $\mathcal{M} \models [\varphi]_{\text{atomic}}$ and since universal properties are downwards closed, $[\mathcal{M}_{\langle\cdot\rangle}]^{\langle\cdot\rangle}_{\text{atomic}} \models [\varphi]_{\text{atomic}}$ and by Lemma 8.2 we have $\mathcal{M}_{\langle\cdot\rangle} \models_{\langle\cdot\rangle} \varphi$. $\square$

Based on this theorem, we repair the synthesized transition system with respect to the corresponding HyperLTL formula. To do so, we enumerate all refinements that leave exactly one transition for each choice. Thereby, the refinement still implements a strategy. Since the HyperLTL formula is a universal, this approach detects a correct refinement iff there is one. Note that also with respect to HyperLTL, this cannot be a complete method, as $\forall^*$ HyperLTL synthesis is undecidable [86]. The general problem of repairing HyperLTL formulas has been discussed in [30], the problem is NP-complete for $\forall^*$ HyperLTL. An approach similar to our has been described for controller synthesis in [31], which distinguishes between controllable and uncontrollable inputs. Implementations have not been developed yet.

8.6 Implementation and Experiments

We implemented a prototype of the repair algorithm as a Python script. Given a synthesized smart contract in form of a Mealy machine and a universal HyperLTL formula, the script checks if the complete system satisfies the property. If not, it searches for free

Table 8.1: Results of the prototype implementation of the repair algorithm. Times are given in seconds. #Calls refers to the number of nuXmv calls that were needed to find a repair.

	Only `vote`		**+ `close`**		**+ owner**		**Full**	
Property	**Time**	**#Calls**	**Time**	**#Calls**	**Time**	**#Calls**	**Time**	**#Calls**
Local Determinism	0.170	1	0.475	1	2.577	1	7.049	1
Local Symmetry	0.229	2	1.251	6	64.90	86	308.1	120
Global No Harm	0.163	1	0.473	1	2.786	1	219.9	86
Determinism	0.147	1	0.612	1	27.63	35	6.825	1
Symmetry	0.254	2	1.630	6	29.03	35	6.571	1
No Harm	0.170	1	0.704	1	2.993	1	90.81	35
Determinism + Symmetry + No Harm	0.274	3	2.105	6	217.7	256	760.4	256

choices and, if there are any, self-composes the system n times for n traces in the quantifier prefix. It then checks if one of the possibilities to resolve the choices satisfies the LTL body of the HyperLTL formula. For LTL model checking, we use the state-of-the-art model checker nuXmv [39]. Using our implementation, we conducted experiments on different versions of the voting contract and on a blinded auction contract.

Voting Contract. We repaired four variants of the voting protocol specified in Specification 8.3 with respect to the hyperproperties described in this chapter. The results are depicted in Table 8.1. None of the three systems initially satisfies either of the properties, the number of calls given in the table therefore refers to the number of refinements tested with nuXmv.

The first variant ("Only `vote`") reduces the contract to the core specification needed for the hyperproperties to make sense. It has a `vote` method and can be implemented as a single-state system. It has two inputs with a free choice; in each case there are two possible transitions. The second variant ("+ `close`") adds the `close` method and initial assumptions, resulting in a three-state system with four choices, again each with two options. The third variant ("+ owner") is the contract as described in Specification 7.1. It has eight choices. This results in 2^8 different combinations that need to be checked.

Lastly, we extended the voting contract with additional features following the voting example of the Solidity Documentation [104]. The extended version additionally records the registered voters in a field `voters`. It also records the addresses that have voted in a field `voted` (similar to →Section 7.1.1). The contract also has two additional →page 84 methods: `giveRightToVote` may be called by the owner of the contract and adds addresses to `voters`; `getWinner` may be called after the voting has been closed to learn which candidate won. The synthesized state machine has again eight choices but is naturally larger than the state machine depicted in Figure 8.1.

Determinism, local determinism, symmetry, and the no harm property refer to the formulas given in Section 8.2. For symmetry, we included the necessary assumption described in Section 8.4.1. The "global no harm" property formulates the no harm property with a single $\square$ instead of $\mathcal{W}$. Local symmetry states symmetry with respect to the greater predicate as well as the `voteA` and `voteB` inputs, similar to local determinism.

Blind Auction. As a second contract, we specified a blind auction in TSL, following [99]. This is the same contract used as benchmark in →Section 7.5.2. Similar to the voting protocol, we had to restrict ourselves to a finite number of bidders, otherwise the hyperproperties would not have been expressible in HyperTSL. The idea of a blind auction is that bidders send a hash of their actual bid and deposit a value which might be higher or lower than the actual bid. After the bidding is closed, bidders reveal their bids. If the hash fits the revealed bid and the deposited value is higher than the actual bid, the bid is valid. The winner of the auction is the bidder with the highest valid bid. Bidders can also withdraw their deposits once a higher bid was revealed. →page 103

We specified the contract with two bidders. We use methods `bidA` and `bidB` for bids as well as `closeBidding`, `revealA`, `revealB`, `closeRevealing`, and `withdraw`. One of the temporal requirements is that bids can only be placed as long as the bidding has not been closed.

$$\square(\texttt{bidA} \lor \texttt{bidB} \rightarrow \boxminus \neg\texttt{closeBidding})$$

The specification contains fields `bidsA`, `bidsB`, `highestBidder`, and `highestBid`. One of the obligations on the fields is to update `highestBid` if bidder `A` reveals a correct bid which is higher than all bids revealed so far.

$$\begin{aligned}&\square(\texttt{revealA} \land \texttt{valid(arg@bid, arg@secret)} \land \texttt{arg@bid} > \texttt{highestBid}\\&\quad \rightarrow [\![\texttt{highestBid} \leftarrowtail \texttt{arg@bid}]\!])\end{aligned}$$

In the above specification, `bid` and `secret` are arguments to the method `revealA`. Similar to the voting case, the specification leaves the free choice which bid is stored as highest bid if the currently revealed bid is the same is the current highest bid. The synthesized state machine leaves free choices at two states. Using our implementation, we obtained strategies to resolve a tie in a way that local determinism or local symmetry are satisfied. For local determinism, the tool needed 1 call to nuXmv and took a running time of 11.13s. For symmetry, it took 2 calls to nuXmv with a total running time of 17.42s.

Evaluation. Our experiments show that the hyperproperties discussed in this chapter are realizable for multiple variants of a voting contract and a blinded auction. The evaluation shows that the runtime mainly depends on the number of choices that had to be tested with nuXmv. The runtime also increases with larger formulas and larger state machines. This implementation is only a prototype built in order to check if the idea to resolve free choices via repair works for relevant hyperproperties. There are many options to improve its performance. For example, one could switch to a model checker that is based on Büchi or Parity automata (e.g., spot [71]). Like this, the LTL formula (which stays the same for every call) would have to be translated to an automaton only once.

Chapter 9

Related Work

Similarly to Part I, we complement the motivating work discussed in the introduction with a more extensive discussion of directly related results. We discuss the related work on the synthesis of smart contracts, how other approaches model the temporal control flow as state machines, and hyperproperties occurring in the context of smart contracts. Furthermore, we compare our technique based on parameters with parameterized synthesis approaches and discuss the use of TSL for synthesis. For more related work on hyperlogics, we refer to →Chapter 5. →page 71

Synthesis of Smart Contracts. Apart from [91], there is only little work so far on reactive synthesis for smart contracts. The major challenge in synthesizing smart contracts is the question of "how to strike a balance between simplicity and expressivity [...] to allow effective synthesis of practical smart contracts" [216]. The authors of [216] propose to apply reactive synthesis from LTL specifications to obtain state machines describing the contract's control flow. The approach does not include any reasoning about the contract's data. The problem of separating data and control flow has been proposed to overcome by splitting contracts into a tuple of protocol, properties, and rules [206]. The protocol is a regular language; properties are given in linear dynamic logic on finite traces (LDL$_f$) and range over method calls and atomic propositions; rules define the reaction to transactions with JavaScript code. The level of formality in the provided guarantees in the above work remains vague, however. Apart from temporal logics, it has been proposed to use deontic logics (logics including obligations like "must" and "may") to express contracts closely to their formulation in natural language [107]. The resulting contracts are an interface rather than executable code and have no means to express the temporal control flow of the smart contract.

Smart Contract Modeling with State Machines. Besides synthesis from logical specifications, another line of work is to model the control flow of a contract with a formal model and translate it to Solidity code. In the FSolidM and VeriSolid frameworks [174, 175], the user generates a graphical representation of a finite state machine, which is translated to Solidity code. The transition labels consist of transactions and guards but do not include access rights. The state logic of smart contracts has also been modeled using Petri nets [235] augmented with guards, which are model-checked for basic errors like deadlocks and then translated to Solidity code. Smart contracts have also been modeled in the Business Process Model and Notation (BPMN) [47, 167] and Unified Modeling Language (UML) [111] standards.

Hyperproperties in Smart Contracts. The analysis of smart contract with respect to hyperproperties is still in its infancy. Most work focuses on analyzing contracts with respect to a specific property. The DAO bug [180], a reentrancy bug that led to the theft of $50 million in 2016, was a turning point in the research on smart contracts. To prevent such bugs in the future, formal properties have been proposed to formalize the desired behavior. To distinguish benign reentrancy from an attack, one needs to compare execution traces. Thus, these properties are often hyperproperties. As an example, call integrity, which formulates that a smart contract may not depend on untrusted code, has been identified to be a $\forall^2$ hyperproperty [117]. Another property is callback freedom [118], which is a $\forall\exists$ hyperproperty [6].

Parameterized Synthesis. Parameterized synthesis has been studied mainly in the context of distributed architectures, in which several components of the system interact with each other. Here, the system is parameterized in the number of the components it contains, e.g., the number of clients [140]. The problem is undecidable in general, even for simpler structures such as token rings, but bounded synthesis approaches have been applied successfully [148, 147]. For the restriction to safety properties, parameterized synthesis has also been tackled by using Angluin's L^* algorithm [172]. Compared with our approach, parameterized synthesis does not have to deal with the implicit hierarchy of our setting, which requires that outputs may only depend on some inputs, depending on the how they are parameterized. Parameterized synthesis is also usually not studied in the context of software.

Temporal Stream Logic (TSL). TSL has been designed specifically for the synthesis of systems with arbitrary data domains. TSL has been used successfully to synthesize larger hardware systems, e.g., a full space shooter arcade game running on an FPGA [112]. First applications for software systems were in the domain of functional programs [92]. The abstraction from the concrete implementation of predicates and functions calls for the combination of reactive synthesis with domain-specific reasoning and SMT solving [90, 171]. Combining TSL with domain-specific reasoning without sacrificing a good performance still remains a challenge. Recently, TSL synthesis has been combined with syntax-guided synthesis, which is used to automatically construct assumptions about functions and predicates [43]. As we have experienced as well, such assumptions are sometimes needed to make a specification realizable. It might therefore be interesting to see if this approach also works in our setting.

Chapter 10

Discussion

In this thesis, we have studied logical methods for reasoning about hyperproperties. We have investigated both sides of the logical medal: the relative expressiveness of different logics and algorithms for the satisfiability and synthesis problems. Regarding the former, we have considered a variety of hyperlogics based on different logical principles. Our expressiveness study shows that quantifier-based temporal hyperlogics and first-order/second-order hyperlogics can be arranged in a true hierarchy of hyperlogics, whereas temporal logics with team semantics provide a fundamentally different perspective on hyperproperties. We also proposed logics tailored for the expression of (hyper)properties in infinite-state systems with data and illustrated their use in the context of smart contracts. Algorithmically, we have shown that while reasoning about hyperproperties is a challenge, solutions can be obtained with careful constructions of expressive logical fragments and suitable approximations. This observation applies to both the satisfiability and the synthesis problem, for which we proposed fragments that build on safety properties and described sound approximations for these generally undecidable problems.

In the following, we first discuss some concluding observations that result from this thesis. Subsequently, we provide an outlook on how to further advance the theory of hyperlogics based on the results of this thesis.

10.1 Conclusions

The Two Dimensions of Hyperproperties. Linear-time hyperproperties describe sets of traces and as such add – compared with linear-time trace properties – a second dimension to the model. The result is a special form of grid: while the time dimension is strictly ordered, the trace dimension is not; the set does not even have to be enumerable in general. The logics we include in our expressiveness study use different types of quantification over this grid, which explains the resulting hierarchy of hyperlogics.

HyperLTL employs first-order quantification on the trace dimension followed by first-order quantification (in form of temporal operators) on the time dimension. FO$[<, E]$ is more expressive as it mixes these two types of quantifiers. HyperQPTL adds another layer of expressiveness by mixing the trace quantifiers with second-order quantification over the time dimension. Finally, HyperQPTL$^+$ features second-order quantification over both dimensions. Our hierarchy thus leaves open certain combinations of quantifiers, e.g., second-order trace quantifiers mixed with first-order time quantifiers. Similar observations have been made in [21] for Hypertrace Logic and for the difference between HyperLTL and FO$[<, E]$ also in [95].

Safety Properties for Logical Reasoning. For both the satisfiability and the synthesis problem, some sort of restriction to safety properties has been the key to obtaining simpler algorithms; for the satisfiability problem in the form of temporal safety and for the synthesis problem in the form of the past-time fragment of TSL. This is not entirely surprising as similar observations have been made for safety properties in various areas [154, 133]. It is interesting, however, that the hyperproperty setting induces various ways to obtain a safety fragment. There is the notion of hypersafety, which is very useful especially for the verification of hyperproperties [77, 213]. For satisfiability, temporal safety proved to be more suitable. In the synthesis setting, we only restricted the trace properties (given as TSL specifications) to the safety fragment and considered arbitrary $\forall^*$ HyperTSL formulas.

Combining Functional Properties and Hyperproperties. To actually synthesize smart contracts from hyperproperties, the main idea was to treat the HyperTSL hyperproperty separately from the TSL trace properties. Before, hyperproperty synthesis has been studied in isolation only [86, 84]. This is valid, of course, since hyperproperties subsume trace properties. Moving some of the algorithmic task to trace property synthesis, however, made it possible to synthesize a solution within very reasonable time. A similar observation has been made in the context of HyperLTL satisfiability: in [25], simple $\forall^*\exists^*$ HyperLTL formulas have been combined with arbitrary LTL formulas. This resulted, among others, in the first decidability result for formulas that can enforce models with infinitely many traces (note that a different part of [25] forms the contribution of Chapter 4).

Algorithms for Expressive Logics. Especially the second part of this thesis showed that algorithms based on expressive logics can be successful, even if the general problems are undecidable. The expressiveness of (parameterized) TSL and HyperTSL allows for a precise expression of software properties, mainly because of the concept of cells, which can be used to model fields. The combination of cells and (uninterpreted) functions and predicates also makes the logics quite readable. The approximations based on LTL and HyperLTL leverage the algorithms available for these logics. That way, we may sacrifice completeness but obtain readable logics that can express the desired functionality, while we are also able to construct solutions in many cases.

10.2 Outlook

Finally, we want to point to open challenges and interesting directions to advance the knowledge about the expressiveness of hyperlogics and associated algorithms.

10.2.1 Expressiveness

The hierarchy presented in Chapter 3 contains a variety of different logics, but such a work can never be considered complete. There are several questions that result from our work.

Epistemic Logics. An immediate first question concerns previously proposed logics whose expressiveness should fall between that of HyperLTL and HyperQPTL$^+$. One class of such logics are epistemic logics, as previously discussed in [56]. Epistemic logics express the knowledge of agents or components acting in distributed systems. One way to obtain such logics is by extending temporal logics with the *knowledge operator* $\mathcal{K}$ [126]. LTL$_\mathcal{K}$ and HyperLTL are known to be of incomparable expressiveness [34], while HyperQPTL subsumes both logics [195]. The proof from [195] can be easily adapted to show that FO$[<, E]$ subsumes LTL$_\mathcal{K}$ and also HyperLTL extended with $\mathcal{K}$. The remaining open question is whether FO$[<, E]$ is strictly more expressive than HyperLTL$_\mathcal{K}$. A possible candidate formula that distinguishes the two logics might be the property "there are two points in time such that all traces repeat their a-value seen at the first position at the second position. The formula is expressible in FO$[<, E]$ as follows:

$$\exists t_1, t_2. \forall x, y. E(t_1, x) \land E(t_2, y) \land x \leq y \rightarrow P_a(x) \leftrightarrow P_a(y)$$

In the formula above, $x \leq y$ enforces that x and y reside on the same trace. A proof that the above formula indeed is not expressible in HyperLTL$_\mathcal{K}$ is still outstanding. One possible first step could be to relate HyperLTL$_\mathcal{K}$ to the linear interpretation of HyperCTL* as defined in [34]. Both logics might be expressively equivalent.

Understanding Team Semantics. Our expressiveness study shows that linear-time logics with teams semantics are hard to compare to quantifier-based linear-time hyperlogics. It seems like we do not yet fully understand the expressiveness of TeamLTL and related logics. For example, we do not know whether TeamLTL (and TeamLTL($\otimes$)) are subsumed by HyperQPTL. A problem that is open since the introduction of TeamLTL in 2018 [153] is the question whether TeamLTL model checking is decidable. This problem is a real challenge due to the split operator $\vee$, which can split the trace set T in any way, possibly resulting in two trace sets that cannot be produced by Kripke structures. Similarly interesting is the relation of branching-time logics with team semantics (as proposed, e.g., in [152, 120]) and logics like HyperCTL* and HyperQCTL*. Here, the major question is whether these logics are similarly hard to compare as their linear-time analogs are.

Beyond our Hierarchy. As a second step, we should expand the hierarchy beyond logics that express hyperproperties over ω-regular words. As discussed in Chapter 5, various types of hyperlogics have been proposed whose expressiveness most likely does not fall between the ones of HyperLTL and HyperQPTL$^+$, for example probabilistic hyperlogics or logics over real-valued streams. Other types like some asynchronous hyperproperties might be expressible in a second-order hyperlogic, e.g., by quantifying stutter-equivalent sets of traces. The challenge here is to find a suitable framework or logic that enables an extensive study of these quite different logics. By understanding the relative expressiveness of these hyperlogics, we could gain knowledge about the conceptual differences and about the complexity of the associated algorithms.

10.2.2 Algorithms

Logical reasoning for hyperproperties remains a challenging task. Based on the experience and observations made in this thesis, we suggest a few directions for further algorithmic investigation.

Simplifying Hyperproperty Reasoning. One of our algorithmic observations is that in the case of hyperproperties, different versions of safety simplify many problems. It would be interesting to reevaluate known algorithmic results for HyperLTL with respect to its hypersafety and especially its temporal safety fragments. For example, synthesis of $\forall^*$ HyperLTL might be simpler when restricting the body to temporal safety. Another algorithmic observation is that keeping trace properties and hyperproperties separate can lead to new algorithms. Hyperproperties usually occur not in isolation but are evaluated with respect to a system that can be described with trace properties. Especially for synthesis, it might be worth to study this approach more generally. One questions is, for example, for which classes of (hyper)properties a system synthesized from trace properties can be refined to satisfy the hyperproperty. The repair approach could also be extended to not only remove transitions from the winning region but also to unroll the system.

Algorithms and Parameters for HyperTSL. The introduction of HyperTSL and HyperTSL$_{\text{rel}}$ has opened many exciting paths for future work. A first question is how to approximate the HyperTSL realizability problem for formulas with quantifier alternations. As the approximations naturally lose information on the content of the cells, it might be challenging to deal with spurious counterexamples in the presence of quantifier alternations. In another direction, Chapter 7 has shown that many smart contracts are best specified using parameters. The obvious next step is to extend HyperTSL with universally quantified parameters. Here, the first question would be in which order the different types of quantifiers may appear in a formula. Lastly, as a way to profit from the expressiveness of HyperTSL$_{\text{rel}}$, one would have to include theories (e.g., the theory of equality) into the synthesis from HyperTSL$_{\text{rel}}$ specifications. Prior work has made steps in that direction for TSL [171, 90], one could extend these ideas to HyperTSL$_{\text{rel}}$. Using theories, HyperTSL$_{\text{rel}}$ could also be interesting in the context of model checking smart contracts or other infinite-state systems.

Bibliography

[1] 2018. *Bamboo: a language for morphing smart contracts.* Retrieved August 12, 2022 from https://github.com/pirapira/bamboo

[2] Erika Ábrahám, Ezio Bartocci, Borzoo Bonakdarpour, and Oyendrila Dobe. 2020. Probabilistic Hyperproperties with Nondeterminism. 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, 518–534. https://doi.org/10.1007/978-3-030-59152-6_29

[3] Erika Ábrahám and Borzoo Bonakdarpour. 2018. HyperPCTL: A Temporal Logic for Probabilistic Hyperproperties. In *Quantitative Evaluation of Systems - 15th International Conference, QEST 2018, Beijing, China, September 4-7, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11024)*, Annabelle McIver and András Horváth (Eds.). Springer, 20–35. https://doi.org/10.1007/978-3-319-99154-2_2

[4] Karl Raymond Abrahamson. 1980. *Decidability and Expressiveness of Logics of Processes.* Ph.D. Dissertation. University of Washington, Seattle, WA, USA.

[5] 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

[6] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming Callbacks for Smart Contract Modularity. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 209:1–209:30. https://doi.org/10.1145/3428277

[7] Mack W. Alford, Leslie Lamport, and Geoff P. Mullery. 1984. Basic Concepts. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course,*

April 3-12, 1984 and April 16-25, 1985, Munich, Germany (Lecture Notes in Computer Science, Vol. 190), Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider (Eds.). Springer, 7–43. https://doi.org/10.1007/3-540-15216-4_12

[8] Bowen Alpern and Fred B. Schneider. 1985. Defining Liveness. *Inf. Process. Lett.* 21, 4 (1985), 181–185. https://doi.org/10.1016/0020-0190(85)90056-0

[9] Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe. 2022. Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts. *CoRR* abs/2205.07529 (2022). https://doi.org/10.48550/arXiv.2205.07529 arXiv:2205.07529

[10] Krzysztof R. Apt and Dexter Kozen. 1986. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.* 22, 6 (1986), 307–309. https://doi.org/10.1016/0020-0190(86)90071-2

[11] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL (2019), 71:1–71:31. https://doi.org/10.1145/3290384

[12] Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. 2017. Hypercollecting Semantics and its Application to Static Analysis of Information Flow. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 874–887. https://doi.org/10.1145/3009837.3009889

[13] Avolabs. 2021. *NFT Auction Contract.* Retrieved August 12, 2022 from https://github.com/avolabs-io/nft-auction

[14] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram Rajamani. 2004. *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft.* Technical Report MSR-TR-2004-08. 22 pages. https://www.microsoft.com/en-us/research/publication/slam-and-static-driver-verifier-technology-transfer-of-formal-methods-inside-microsoft/

[15] Musard Balliu, Mads Dam, and Gurvan Le Guernic. 2011. Epistemic Temporal Logic for Information Flow Security. In *Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011, San Jose,*

CA, USA, 5 June, 2011, Aslan Askarov and Joshua D. Guttman (Eds.). ACM, 6. https://doi.org/10.1145/2166956.2166962

[16] Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. 2021. Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. *Department of Computer Science Report Series B, vol. B-2021-1, Department of Computer Science, University of Helsinki* (2021). https://helda.helsinki.fi/handle/10138/333647

[17] Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. 2021. *The Results of SAT Competition 2021.* Retrieved August 12, 2022 from https://satcompetition.github.io/2021/slides/ISC2021-fixed.pdf

[18] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, 100–114. https://doi.org/10.1109/CSFW.2004.17

[19] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 90–101. https://doi.org/10.1145/1480881.1480894

[20] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 97–110. https://doi.org/10.1145/2103656.2103670

[21] Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. 2022. Flavors of Sequential Information Flow. In *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13182)*, Bernd Finkbeiner and Thomas Wies (Eds.). Springer, 1–19. https://doi.org/10.1007/978-3-030-94583-1_1

[22] Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. 2021. A Temporal Logic for Asynchronous Hyperproperties. In

Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759), Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 694–717. https://doi.org/10.1007/978-3-030-81685-8_33

[23] Bernhard Beckert and Mattias Ulbrich. 2018. Trends in Relational Program Verification. In *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, Peter Müller and Ina Schaefer (Eds.). Springer, 41–58. https://doi.org/10.1007/978-3-319-98047-8_3

[24] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 14–25. https://doi.org/10.1145/964001.964003

[25] Raven Beutner, David Carral, Bernd Finkbeiner, Jana Hofmann, and Markus Krötzsch. 2022. Deciding Hyperproperties Combined with Functional Specifications. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 56:1–56:13. https://doi.org/10.1145/3531130.3533369

[26] Raven Beutner and Bernd Finkbeiner. 2021. A Temporal Logic for Strategic Hyperproperties. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference (LIPIcs, Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:19. https://doi.org/10.4230/LIPIcs.CONCUR.2021.24

[27] Raven Beutner and Bernd Finkbeiner. 2022. Software Verification of Hyperproperties Beyond k-Safety. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 341–362. https://doi.org/10.1007/978-3-031-13185-1_17

[28] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, Toby C. Murray and Deian Stefan (Eds.). ACM, 91–96. https://doi.org/10.1145/2993600.2993611

[29] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded Model Checking. *Adv. Comput.* 58 (2003), 117–148. https://doi.org/10.1016/S0065-2458(03)58003-2

[30] 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)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer, 423–441. https://doi.org/10.1007/978-3-030-31784-3_25

[31] Borzoo Bonakdarpour and Bernd Finkbeiner. 2020. Controller Synthesis for Hyperproperties. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, 366–379. https://doi.org/10.1109/CSF49147.2020.00033

[32] Borzoo Bonakdarpour, Pavithra Prabhakar, and César Sánchez. 2020. Model Checking Timed Hyperproperties in Discrete-Time Systems. In *NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12229)*, Ritchie Lee, Susmit Jha, and Anastasia Mavridou (Eds.). Springer, 311–328. https://doi.org/10.1007/978-3-030-55754-6_18

[33] Borzoo Bonakdarpour and Sarai Sheinvald. 2021. Finite-Word Hyperlanguages. In *Language and Automata Theory and Applications - 15th International Conference, LATA 2021, Milan, Italy, March 1-5, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12638)*, Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron (Eds.). Springer, 173–186. https://doi.org/10.1007/978-3-030-68195-1_17

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

[35] Laura Bozzelli, Adriano Peron, and César Sánchez. 2021. Asynchronous Extensions of HyperLTL. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470583

[36] J. Richard Büchi. 1960. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly* 6 (1960), 66–92. https://doi.org/10.1002/malq.19600060105

[37] J. Richard Büchi and Lawrence H. Landweber. 1969. Solving Sequential Conditions by Finite-State Strategies. *Trans. Amer. Math. Soc.* 138 (1969), 295–311. http://www.jstor.org/stable/1994916

[38] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1990. Symbolic Model Checking: 10ˆ20 States and Beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990.* IEEE Computer Society, 428–439. https://doi.org/10.1109/LICS.1990.113767

[39] 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)*, Armin Biere and Roderick Bloem (Eds.). Springer, 334–342. https://doi.org/10.1007/978-3-319-08867-9_22

[40] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. 2020. Securing Smart Contracts with Information Flow. In *International Symposium on Foundations and Applications of Blockchain (FAB).*

[41] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. 2021. Compositional Security for Reentrant Applications. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021.* IEEE, 1249–1267. https://doi.org/10.1109/SP40001.2021.00084

[42] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 875–890. https://doi.org/10.1145/3133956.3134058

[43] Wonhyuk Choi, Bernd Finkbeiner, Ruzica Piskac, and Mark Santolucito. 2022. Can Reactive Synthesis and Syntax-Guided Synthesis Be Friends?. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 229–243. https://doi.org/10.1145/3519939.3523429

[44] Stephen Chong, K. Vikram, and Andrew C. Myers. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, Niels Provos (Ed.). USENIX Association. https://www.usenix.org/conference/16th-usenix-security-symposium/sif-enforcing-confidentiality-and-integrity-web

[45] Andrey Chudnov, George Kuan, and David A. Naumann. 2014. Information Flow Monitoring as Abstract Interpretation for Relational Logic. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society, 48–62. https://doi.org/10.1109/CSF.2014.12

[46] Alonzo Church. 1957. Applications of Recursive Arithmetic to the Problem of Circuit Synthesis. In *Summaries of the Summer Institute of Symbolic Logic, Cornell University, Ithaca, NY*. 3–50. https://doi.org/10.2307/2271310

[47] Claudio Di Ciccio, Alessio Cecconi, Marlon Dumas, Luciano García-Bañuelos, Orlenys López-Pintado, Qinghua Lu, Jan Mendling, Alexander Ponomarev, An Binh Tran, and Ingo Weber. 2019. Blockchain Support for Collaborative Business Processes. *Inform. Spektrum* 42, 3 (2019), 182–190. https://doi.org/10.1007/s00287-019-01178-x

[48] Alessandro Cimatti, Marco Roveri, and Daniel Sheridan. 2004. Bounded Verification of Past LTL. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3312)*, Alan J. Hu and Andrew K. Martin (Eds.). Springer, 245–259. https://doi.org/10.1007/978-3-540-30494-4_18

[49] Edmund M. Clarke. 2008. The Birth of Model Checking. In *25 Years of Model Checking - History, Achievements, Perspectives (Lecture Notes in Computer Science, Vol. 5000)*, Orna Grumberg and Helmut Veith (Eds.). Springer, 1–26. https://doi.org/10.1007/978-3-540-69850-0_1

[50] 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*. 265–284. https://doi.org/10.1007/978-3-642-54792-8_15

[51] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. IEEE Computer Society, 51–65. https://doi.org/10.1109/CSF.2008.7

[52] Michael J. Coblenz. 2017. Obsidian: A Safer Blockchain Programming Language. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 97–99. https://doi.org/10.1109/ICSE-C.2017.150

[53] Norine Coenen, Raimund Dachselt, Bernd Finkbeiner, Hadar Frenkel, Christopher Hahn, Tom Horak, Niklas Metzger, and Julian Siber. 2022. Explaining Hyperproperty Violations. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 407–429. https://doi.org/10.1007/978-3-031-13185-1_20

[54] Norine Coenen, Bernd Finkbeiner, Hadar Frenkel, Christopher Hahn, Niklas Metzger, and Julian Siber. 2022. Temporal Causality in Reactive Systems. In *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13505)*, Ahmed Bouajjani, Lukás Holík, and Zhilin Wu (Eds.). Springer, 208–224. https://doi.org/10.1007/978-3-031-19992-9_13

[55] 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

[56] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. 2020. *The Hierarchy of Hyperlogics: A Knowledge Reasoning Perspective.* Retrieved August 12, 2022 from https://www.react.uni-saarland.de/publications/CFHH20.html

[57] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Yannick Schillo. 2021. Runtime Enforcement of Hyperproperties. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12971)*, Zhe Hou and Vijay Ganesh (Eds.). Springer, 283–299. https://doi.org/10.1007/978-3-030-88885-5_19

[58] Norine Coenen, Bernd Finkbeiner, Jana Hofmann, and Julia Tillman. 2022. Smart Contract Synthesis Modulo Hyperproperties. https://doi.org/10.48550/ARXIV.2208.07180 To appear at the 36th IEEE Computer Security Foundations Symposium (CSF 2023).

[59] 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)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 121–139. https://doi.org/10.1007/978-3-030-25540-4_7

[60] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman (Eds.). ACM, 151–158. https://doi.org/10.1145/800157.805047

[61] Jukka Corander, Antti Hyttinen, Juha Kontinen, Johan Pensar, and Jouko Väänänen. 2019. A Logical Approach to Context-Specific Independence. *Ann. Pure Appl. Logic* 170, 9 (2019), 975–992. https://doi.org/10.1016/j.apal.2019.04.004

[62] Pedro R. D'Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. 2017. Is Your Software on Dope? - Formal Analysis of Surreptitiously "enhanced" Programs. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 83–110. https://doi.org/10.1007/978-3-662-54434-1_4

[63] Ádám Darvas, Reiner Hähnle, and David Sands. 2005. A Theorem Proving Approach to Analysis of Secure Information Flow. In *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3450)*, Dieter Hutter and Markus Ullmann (Eds.). Springer, 193–209. https://doi.org/10.1007/978-3-540-32004-3_20

[64] Luca de Alfaro, Thomas A. Henzinger, and Orna Kupferman. 1998. Concurrent Reachability Games. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*. IEEE Computer Society, 564–575. https://doi.org/10.1109/SFCS.1998.743507

[65] 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

[66] Samvid Dharanikota, Suvam Mukherjee, Chandrika Bhardwaj, Aseem Rastogi, and Akash Lal. 2021. Celestial: A Smart Contracts Verification Framework. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 133–142. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_22

[67] 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

[68] Goran Doychev and Boris Köpf. 2017. Rigorous Analysis of Software Countermeasures against Cache Attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 406–421. https://doi.org/10.1145/3062341.3062388

[69] Geir Dullerud and Fernando Paganini. 2000. *A Course in Robust Control Theory*. Springer New York, NY. https://doi.org/10.1007/978-1-4757-3290-0

[70] Arnaud Durand, Miika Hannula, Juha Kontinen, Arne Meier, and Jonni Virtema. 2018. Probabilistic Team Semantics. In *FoIKS (Lecture Notes in Computer Science, Vol. 10833)*. Springer, 186–206. https://doi.org/10.1007/978-3-319-90050-6_11

[71] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 - A Framework for LTL and Omega-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)*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). 122–129. https://doi.org/10.1007/978-3-319-46520-3_8

[72] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography,*

Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3876), Shai Halevi and Tal Rabin (Eds.). Springer, 265–284. https://doi.org/10.1007/11681878_14

[73] Rüdiger Ehlers. 2010. Symbolic Bounded Synthesis. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 365–379. https://doi.org/10.1007/978-3-642-14295-6_33

[74] 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

[75] Anna Ernst. 2020. *Polizei ermittelt nach Hacker-Angriff in einem Todesfall.* Retrieved August 12, 2022 from https://www.sueddeutsche.de/panorama/duesseldorf-uniklinikum-erpressung-hacker-angriff-1.5035140

[76] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. 1995. *Reasoning About Knowledge.* MIT Press. https://doi.org/10.7551/mitpress/5803.001.0001

[77] Azadeh Farzan and Anthony Vandikas. 2019. Automated Hypersafety Verification. 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)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 200–218. https://doi.org/10.1007/978-3-030-25540-4_11

[78] 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)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 325–332. https://doi.org/10.1007/978-3-319-63390-9_17

[79] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. 2009. An Antichain Algorithm for LTL Realizability. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 263–277. https://doi.org/10.1007/978-3-642-02658-4_22

[80] Bernd Finkbeiner. 2016. Synthesis of Reactive Systems. In *Dependable Software Systems Engineering*, Javier Esparza, Orna Grumberg, and Salomon Sickert

(Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 45. IOS Press, 72–98. https://doi.org/10.3233/978-1-61499-627-9-72

[81] Bernd Finkbeiner, Lennart Haas, and Hazem Torfah. 2019. Canonical Representations of k-Safety Hyperproperties. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 17–31. https://doi.org/10.1109/CSF.2019.00009

[82] 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)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:14. https://doi.org/10.4230/LIPIcs.CONCUR.2016.13

[83] 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)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 521–527. https://doi.org/10.1007/978-3-030-01090-4_31

[84] Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Leander Tentrup. 2020. Realizing Omega-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)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 40–63. https://doi.org/10.1007/978-3-030-53291-8_4

[85] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. 2018. Synthesizing Reactive Systems from Hyperproperties. In *Proceedings of CAV (LNCS, Vol. 10981)*. Springer, 289–306. https://doi.org/10.1007/978-3-319-96145-3_16

[86] 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

[87] 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), Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 564–570. https://doi.org/10.1007/978-3-319-63390-9_29

[88] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2017. Monitoring Hyperproperties. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10548)*, Shuvendu K. Lahiri and Giles Reger (Eds.). Springer, 190–207. https://doi.org/10.1007/978-3-319-67531-2_12

[89] 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)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 144–163. https://doi.org/10.1007/978-3-319-96145-3_8

[90] Bernd Finkbeiner, Philippe Heim, and Noemi Passing. 2022. Temporal Stream Logic modulo Theories. In *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13242)*, Patricia Bouyer and Lutz Schröder (Eds.). Springer, 325–346. https://doi.org/10.1007/978-3-030-99253-8_17

[91] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. 2022. Reactive Synthesis of Smart Contract Control Flows. https://doi.org/10.48550/ARXIV.2205.06039

[92] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Synthesizing Functional Reactive Programs. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, Richard A. Eisenberg (Ed.). ACM, 162–175. https://doi.org/10.1145/3331545.3342601

[93] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Temporal Stream Logic: Synthesis Beyond the Bools. 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)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 609–629. https://doi.org/10.1007/978-3-030-25540-4_35

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

[95] 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)*, Heribert Vollmer and Brigitte Vallée (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:14. https://doi.org/10.4230/LIPIcs.STACS.2017.30

[96] Michael J. Fischer and Richard E. Ladner. 1979. Propositional Dynamic Logic of Regular Programs. *J. Comput. Syst. Sci.* 18, 2 (1979), 194–211. https://doi.org/10.1016/0022-0000(79)90046-1

[97] Limor Fix. 2008. *Fifteen Years of Formal Property Verification in Intel.* Springer Berlin Heidelberg, Berlin, Heidelberg, 139–144. https://doi.org/10.1007/978-3-540-69850-0_8

[98] 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. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia (LIPIcs, Vol. 202)*, Filippo Bonchi and Simon J. Puglisi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 47:1–47:19. https://doi.org/10.4230/LIPIcs.MFCS.2021.47

[99] Ethereum Foundation. 2019. *Blind Auction Contract.* Retrieved August 12, 2022 from https://docs.soliditylang.org/en/v0.8.16/solidity-by-example.html#blind-auction

[100] Ethereum Foundation. 2021. *Simple Auction Contract.* Retrieved August 12, 2022 from https://docs.soliditylang.org/en/v0.6.8/solidity-by-example.html#simple-open-auction

[101] Ethereum Foundation. 2022. *Introduction to Ethereum.* Retrieved August 12, 2022 from https://ethereum.org/en/developers/docs/intro-to-ethereum/

[102] Ethereum Foundation. 2022. *Solidity Documentation.* Retrieved August 12, 2022 from https://docs.soliditylang.org/en/v0.8.16/

[103] Fourswords. 2022. *Simple Auction Contract.* Retrieved August 12, 2022 from https://fourswords.io/docs/smartcontracts/solidity-guide/solidity-examples/

[104] Fourswords. 2022. *Voting Contract.* Retrieved August 12, 2022 from https://fourswords.io/docs/smartcontracts/solidity-guide/solidity-examples/

[105] Anthony C. J. Fox, Michael J. C. Gordon, and Magnus O. Myreen. 2010. Specification and Verification of ARM Hardware and Software. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, David S. Hardin (Ed.). Springer, 221–247. https://doi.org/10.1007/978-1-4419-1539-9_8

[106] Nissim Francez. 1983. Product Properties and Their Direct Verification. *Acta Informatica* 20 (1983), 329–344. https://doi.org/10.1007/BF00264278

[107] Christopher Frantz and Mariusz Nowostawski. 2016. From Institutions to Code: Towards Automated Generation of Smart Contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), Augsburg, Germany, September 12-16, 2016*, Sameh Elnikety, Peter R. Lewis, and Christian Müller-Schloer (Eds.). IEEE, 210–215. https://doi.org/10.1109/FAS-W.2016.53

[108] Tim French. 2001. Decidability of Quantifed Propositional Branching Time Logics. In *AI 2001: Advances in Artificial Intelligence, 14th Australian Joint Conference on Artificial Intelligence, Adelaide, Australia, December 10-14, 2001, Proceedings.* 165–176. https://doi.org/10.1007/3-540-45656-2_15

[109] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. 1980. On the Temporal Analysis of Fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Las Vegas, Nevada) *(POPL '80).* ACM, New York, NY, USA, 163–173. https://doi.org/10.1145/567446.567462

[110] Pietro Galliani and Lauri Hella. 2013. Inclusion Logic and Fixed Point Logic. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy (LIPIcs, Vol. 23)*, Simona Ronchi Della Rocca (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 281–295. https://doi.org/10.4230/LIPIcs.CSL.2013.281

[111] Peter Garamvolgyi, Imre Kocsis, Benjamin Gehl, and Attila Klenik. 2018. Towards Model-Driven Engineering of Smart Contracts for Cyber-Physical Systems. In *48th Annual IEEE/IFIP International Conference on Dependable Systems*

and Networks Workshops, DSN Workshops 2018, Luxembourg, June 25-28, 2018. IEEE Computer Society, 134–139. https://doi.org/10.1109/DSN-W.2018.00052

[112] Gideon Geier, Philippe Heim, Felix Klein, and Bernd Finkbeiner. 2019. Syntroids: Synthesizing a Game for FPGAs using Temporal Logic Specifications. In *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, Clark W. Barrett and Jin Yang (Eds.). IEEE, 138–146. https://doi.org/10.23919/FMCAD.2019.8894261

[113] Malik Ghallab, Dana S. Nau, and Paolo Traverso. 2004. *Automated Planning: Theory and Practice*. Elsevier. https://doi.org/10.1016/B978-1-55860-856-6.X5000-5

[114] Giuseppe De Giacomo, Paolo Felli, Marco Montali, and Giuseppe Perelli. 2021. HyperLDLf: a Logic for Checking Properties of Finite Traces Process Logs. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Zhi-Hua Zhou (Ed.). ijcai.org, 1859–1865. https://doi.org/10.24963/ijcai.2021/256

[115] Kurt Gödel. 1930. Die Vollständigkeit der Axiome des Logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik* 37, 1 (1930), 349–360. https://doi.org/10.1007/BF01696781

[116] 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*. 11–20. https://doi.org/10.1109/SP.1982.10014

[117] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10804)*, Lujo Bauer and Ralf Küsters (Eds.). Springer, 243–269. https://doi.org/10.1007/978-3-319-89722-6_10

[118] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to smart contracts. *Proc. ACM Program. Lang.* 2, POPL (2018), 48:1–48:28. https://doi.org/10.1145/3158136

[119] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1–19. https://doi.org/10.1109/SP40000.2020.00011

[120] Jens Oliver Gutsfeld, Arne Meier, Christoph Ohrem, and Jonni Virtema. 2022. Temporal Team Semantics Revisited. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 44:1–44:13. https://doi.org/10.1145/3531130.3533360

[121] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. 2020. Propositional Dynamic Logic for Hyperproperties. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference) (LIPIcs, Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 50:1–50:22. https://doi.org/10.4230/LIPIcs.CONCUR.2020.50

[122] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. 2021. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434319

[123] Christopher Hahn. 2021. *Logical and Deep Learning Methods for Temporal Reasoning*. Ph.D. Dissertation. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/32183

[124] Ákos Hajdu and Dejan Jovanovic. 2020. SMT-Friendly Formalization of the Solidity Memory Model. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 224–250. https://doi.org/10.1007/978-3-030-44914-8_9

[125] Joseph Y. Halpern and Yoram Moses. 1984. Knowledge and Common Knowledge in a Distributed Environment. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984*, Tiko Kameda, Jayadev Misra, Joseph G. Peters, and Nicola Santoro (Eds.). ACM, 50–61. https://doi.org/10.1145/800222.806735

[126] 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

[127] Christian Hammer and Gregor Snelting. 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.* 8, 6 (2009), 399–422. https://doi.org/10.1007/s10207-009-0086-1

[128] Miika Hannula and Juha Kontinen. 2016. A Finite Axiomatization of Conditional Independence and Inclusion Dependencies. *Inf. Comput.* 249 (2016), 121–137. https://doi.org/10.1016/j.ic.2016.04.001

[129] Miika Hannula, Juha Kontinen, Jan Van den Bussche, and Jonni Virtema. 2020. Descriptive Complexity of Real Computation and Probabilistic Independence Logic. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). ACM, 550–563. https://doi.org/10.1145/3373718.3394773

[130] Miika Hannula, Juha Kontinen, Jonni Virtema, and Heribert Vollmer. 2018. Complexity of Propositional Logics in Team Semantic. *ACM Trans. Comput. Log.* 19, 1 (2018), 2:1–2:14. https://doi.org/10.1145/3157054

[131] 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)*, Krzysztof R. Apt (Ed.). Springer, 477–498. https://doi.org/10.1007/978-3-642-82453-1_17

[132] Klaus Havelund and Grigore Rosu. 2002. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2280)*, Joost-Pieter Katoen and Perdita Stevens (Eds.). Springer, 342–356. https://doi.org/10.1007/3-540-46002-0_24

[133] Klaus Havelund and Grigore Rosu. 2004. Efficient Monitoring of Safety Properties. *Int. J. Softw. Tools Technol. Transf.* 6, 2 (2004), 158–173. https://doi.org/10.1007/s10009-003-0117-6

[134] Lauri Hella, Antti Kuusisto, Arne Meier, and Jonni Virtema. 2019. Model checking and Validity in Propositional and Modal Inclusion Logics. *J. Log. Comput.* 29, 5 (2019), 605–630. https://doi.org/10.1093/logcom/exz008

[135] Lauri Hella, Kerkko Luosto, Katsuhiko Sano, and Jonni Virtema. 2014. The Expressive Power of Modal Dependence Logic. In *Advances in Modal Logic*

10, invited and contributed papers from the tenth conference on "Advances in Modal Logic," held in Groningen, The Netherlands, August 5-8, 2014, Rajeev Goré, Barteld P. Kooi, and Agi Kurucz (Eds.). College Publications, 294–312. http://www.aiml.net/volumes/volume10/Hella-Luosto-Sano-Virtema.pdf

[136] Hsi-Ming Ho, Ruoyu Zhou, and Timothy M. Jones. 2019. On Verifying Timed Hyperproperties. In *26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16-19, 2019, Málaga, Spain (LIPIcs, Vol. 147)*, Johann Gamper, Sophie Pinchinat, and Guido Sciavicco (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:18. https://doi.org/10.4230/LIPIcs.TIME.2019.20

[137] Wilfrid Hodges. 1997. Compositional Semantics for a Language of Imperfect Information. *Log. J. IGPL* 5, 4 (1997), 539–563. https://doi.org/10.1093/jigpal/5.4.539

[138] 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)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 94–112. https://doi.org/10.1007/978-3-030-72016-2_6

[139] Tapani Hyttinen, Gianluca Paolini, and Jouko Väänänen. 2017. A Logic for Arguing About Probabilities in Measure Teams. *Arch. Math. Logic* 56, 5-6 (2017), 475–489. https://doi.org/10.1007/s00153-017-0535-x

[140] Swen Jacobs and Roderick Bloem. 2012. Parameterized Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7214)*, Cormac Flanagan and Barbara König (Eds.). Springer, 362–376. https://doi.org/10.1007/978-3-642-28756-5_25

[141] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. 2020. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1695–1712. https://doi.org/10.1109/SP40000.2020.00066

[142] Robert B. Jones, John W. O'Leary, Carl-Johan H. Seger, Mark D. Aagaard, and Thomas F. Melham. 2001. Practical Formal Verification in Microprocessor Design. *IEEE Des. Test Comput.* 18, 4 (2001), 16–25. https://doi.org/10.1109/54.936245

[143] Roope Kaivola. 1997. *Using Automata to Characterise Fixed Point Temporal Logics.* Ph.D. Dissertation. University of Edinburgh. College of Science and Engineering. School of Informatics.

[144] Hans W. Kamp. 1968. *Tense Logic and the Theory of Linear Order.* Ph.D. Dissertation. Computer Science Department, University of California at Los Angeles, USA.

[145] Thomas Kaplan, Ian Austen, and Selam Gebrekidan. 2019. *Boeing Planes Are Grounded in U.S. After Days of Pressure.* Retrieved August 12, 2022 from https://www.nytimes.com/2019/03/13/business/canada-737-max.html

[146] 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.* 2–12. https://doi.org/10.1109/LICS.1995.523239

[147] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. 2013. PARTY Parameterized Synthesis of Token Rings. 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, 928–933. https://doi.org/10.1007/978-3-642-39799-8_66

[148] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. 2013. Towards Efficient Parameterized Synthesis. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7737)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 108–127. https://doi.org/10.1007/978-3-642-35873-9_9

[149] 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

[150] Juha Kontinen, Julian-Steffen Müller, Henning Schnoor, and Heribert Vollmer. 2015. A Van Benthem Theorem for Modal Team Semantics. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany (LIPIcs, Vol. 41)*, Stephan Kreutzer (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 277–291. https://doi.org/10.4230/LIPIcs.CSL.2015.277

[151] Juha Kontinen and Max Sandström. 2021. On the Expressive Power of TeamLTL and First-Order Team Logic over Hyperproperties. In *Logic, Language, Information, and Computation - 27th International Workshop, WoLLIC 2021, Virtual Event, October 5-8, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13038)*, Alexandra Silva, Renata Wassermann, and Ruy J. G. B. de Queiroz (Eds.). Springer, 302–318. https://doi.org/10.1007/978-3-030-88853-4_19

[152] Andreas Krebs, Arne Meier, and Jonni Virtema. 2015. A Team Based Variant of CTL. In *22nd International Symposium on Temporal Representation and Reasoning, TIME 2015, Kassel, Germany, September 23-25, 2015*, Fabio Grandi, Martin Lange, and Alessio Lomuscio (Eds.). IEEE Computer Society, 140–149. https://doi.org/10.1109/TIME.2015.11

[153] 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) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 117)*, Igor Potapov, Paul Spirakis, and James Worrell (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:16. https://doi.org/10.4230/LIPIcs.MFCS.2018.10

[154] 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

[155] Robert P. Kurshan. 2008. *Verification Technology Transfer.* Springer Berlin Heidelberg, Berlin, Heidelberg. 46–64 pages. https://doi.org/10.1007/978-3-540-69850-0_3

[156] Antti Kuusisto. 2015. A Double Team Semantics for Generalized Quantifiers. *Journal of Logic, Language and Information* 24, 2 (2015), 149–191. https://doi.org/10.1007/s10849-015-9217-4

[157] Shuvendu K. Lahiri, Shuo Chen, Yuepeng Wang, and Isil Dillig. 2018. Formal Specification and Verification of Smart Contracts for Azure Blockchain. *CoRR* abs/1812.08829 (2018). arXiv:1812.08829 http://arxiv.org/abs/1812.08829

[158] Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.* 3, 2 (1977), 125–143. https://doi.org/10.1109/TSE.1977.229904

[159] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley. http://research.microsoft.com/users/lamport/tla/book.html

[160] Leslie Lamport. 2019. *Industrial Use of TLA+.* Retrieved August 12, 2022 from https://lamport.azurewebsites.net/tla/industrial-use.html?unhideBut=hide-amazon&unhideDiv=amazon

[161] Frederic Lardinois. 2017. *With Cosmos DB, Microsoft wants to build one database to rule them all.* Retrieved August 12, 2022 from https://techcrunch.com/2017/05/10/with-cosmos-db-microsoft-wants-to-build-one-database-to-rule-them-all/

[162] François Laroussinie and Nicolas Markey. 2014. Quantified CTL: Expressiveness and Complexity. *Logical Methods in Computer Science* 10, 4 (2014). https://doi.org/10.2168/LMCS-10(4:17)2014

[163] Charlie Lee. 2017. *Hackers Seize $32 Million in Ethereum in Parity Wallet Breach.* Retrieved August 12, 2022 from https://www.ccn.com/hackers-seize-32-million-in-parity-wallet-breach/

[164] Czestaw Lejewski. 1959. Time and Modality, By A. N. Prior, Clarendon Press: Oxford University Press, 1957. Pp. viii 148. *Philosophy* 34, 128 (1959), 56–59. https://doi.org/10.1017/S0031819100029776

[165] Leonid Anatolevich Levin. 1973. Universal Sequential Search Problems. *Problems of Information Transmission* 9, 3 (1973).

[166] 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*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[167] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. 2019. Caterpillar: A Business Process Execution Engine on the Ethereum Blockchain. *Softw. Pract. Exp.* 49, 7 (2019), 1162–1193. https://doi.org/10.1002/spe.2702

[168] Martin Lück. 2020. On the Complexity of Linear Temporal Logic with Team Semantics. *Theor. Comput. Sci.* 837 (2020), 1–25. https://doi.org/10.1016/j.tcs.2020.04.019

[169] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. 2020. Practical Synthesis of Reactive Systems from LTL Specifications via Parity Games. *Acta Informatica* 57, 1-2 (2020), 3–36. https://doi.org/10.1007/s00236-019-00349-3

[170] Inês Lynce and João Marques-Silva. 2006. Efficient Haplotype Inference with Boolean Satisfiability. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 104–109. http://www.aaai.org/Library/AAAI/2006/aaai06-017.php

[171] Benedikt Maderbacher and Roderick Bloem. 2021. Reactive Synthesis Modulo Theories Using Abstraction Refinement. *CoRR* abs/2108.00090 (2021). arXiv:2108.00090 https://arxiv.org/abs/2108.00090

[172] Oliver Markgraf, Chih-Duo Hong, Anthony W. Lin, Muhammad Najib, and Daniel Neider. 2020. Parameterized Synthesis with Safety Properties. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, 273–292. https://doi.org/10.1007/978-3-030-64437-6_14

[173] 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)*, Maribel Fernández and Anca Muscholl (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:16. https://doi.org/10.4230/LIPIcs.CSL.2020.29

[174] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers (Lecture Notes in Computer*

Science, Vol. 10957), Sarah Meiklejohn and Kazue Sako (Eds.). Springer, 523–540. https://doi.org/10.1007/978-3-662-58387-6_28

[175] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11598)*, Ian Goldberg and Tyler Moore (Eds.). Springer, 446–465. https://doi.org/10.1007/978-3-030-32101-7_27

[176] Daryl McCullough. 1988. Noninterference and the Composability of Security Properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 18-21, 1988*. IEEE Computer Society, 177–186. https://doi.org/10.1109/SECPRI.1988.8110

[177] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[178] Robert McNaughton and Seymour A. Papert. 1971. *Counter-Free Automata (M.I.T. Research Monograph No. 65)*. The MIT Press.

[179] 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

[180] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. 2019. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *J. Cases Inf. Technol.* 21, 1 (2019), 19–32. https://doi.org/10.4018/JCIT.2019010102

[181] Marvin L. Minsky. 1967. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA.

[182] Faron Moller and Alexander Moshe Rabinovich. 1999. On the Expressive Power of CTL. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 360–368. https://doi.org/10.1109/LICS.1999.782631

[183] David A. Naumann. 2020. Thirty-Seven Years of Relational Hoare Logic: Remarks on Its Principles and History. In *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12477)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 93–116. https://doi.org/10.1007/978-3-030-61470-6_7

[184] Zeinab Nehai, Pierre-Yves Piriou, and Frédéric F. Daumas. 2018. Model-Checking of Smart Contracts. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*. IEEE, 980–987. https://doi.org/10.1109/Cybermatics_2018.2018.00185

[185] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73. https://doi.org/10.1145/2699417

[186] Open Zeppelin. 2022. *ERC20 Token System Contract.* Retrieved August 12, 2022 from https://github.com/OpenZeppelin/openzeppelin-contracts/blob/9b3710465583284b8c4c5d2245749246bb2e0094/contracts/token/ERC20/ERC20.sol

[187] Open Zeppelin. 2022. *ERC20 Token System Documentation.* Retrieved August 12, 2022 from https://docs.openzeppelin.com/contracts/2.x/api/token/erc20

[188] Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy, S&P 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

[189] 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)*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer, 364–380. https://doi.org/10.1007/11609773_24

[190] 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*. 46–57. https://doi.org/10.1109/SFCS.1977.32

[191] 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

[192] Rob Price. 2016. *Digital currency Ethereum is cratering because of a $50 million hack*. Retrieved August 12, 2022 from https://www.businessinsider.com/dao-hacked-ethereum-crashing-in-value-tens-of-millions-allegedly-stolen-2016-6

[193] Program the Blockchain. 2022. *Coin Toss Contract*. Retrieved August 12, 2022 from https://programtheblockchain.com/posts/2018/03/16/flipping-a-coin-in-ethereum/

[194] Program the Blockchain. 2022. *Crowdfunding Contract*. Retrieved August 12, 2022 from https://programtheblockchain.com/posts/2018/01/19/writing-a-crowdfunding-contract-a-la-kickstarter/

[195] Markus N. Rabe. 2016. *A Temporal Logic Approach to Information-Flow Control*. Ph.D. Dissertation. Saarland University. http://scidok.sulb.uni-saarland.de/volltexte/2016/6387/

[196] Michael O. Rabin and Dana S. Scott. 1959. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.* 3, 2 (1959), 114–125. https://doi.org/10.1147/rd.32.0114

[197] J. Richard Büchi. 1966. Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic. In *Logic, Methodology and Philosophy of Science*, Ernest Nagel, Patrick Suppes, and Alfred Tarski (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 44. Elsevier, 1–11. https://doi.org/10.1016/S0049-237X(09)70564-6

[198] John Alan Robinson and Andrei Voronkov (Eds.). 2001. *Handbook of Automated Reasoning*. Elsevier and MIT Press. https://doi.org/10.1016/B978-044450813-3/50000-X

[199] Hartley Rogers. 1987. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA.

[200] David M. Russinoff. 1998. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division and Square Root Algorithms of the AMD-K7™ Processor. *LMS Journal of Computation and Mathematics* 1 (1998), 148–200. https://doi.org/10.1112/S1461157000000176

[201] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society, 186–199. https://doi.org/10.1109/CSF.2010.20

[202] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and Principles. *J. Comput. Secur.* 17, 5 (2009), 517–548. https://doi.org/10.3233/JCS-2009-0352

[203] 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

[204] Shubham Sahai, Pramod Subramanyan, and Rohit Sinha. 2020. Verification of Quantitative Hyperproperties Using Trace Enumeration Relations. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 201–224. https://doi.org/10.1007/978-3-030-53288-8_11

[205] Katsuhiko Sano and Jonni Virtema. 2019. Characterising Modal Definability of Team-Based Logics via the Universal Modality. *Ann. Pure Appl. Log.* 170, 9 (2019), 1100–1127. https://doi.org/10.1016/j.apal.2019.04.009

[206] Naoto Sato, Takaaki Tateishi, and Shunichi Amano. 2018. Formal Requirement Enforcement on Smart Contracts Based on Linear Dynamic Logic. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*. IEEE, 945–954. https://doi.org/10.1109/Cybermatics_2018.2018.00181

[207] Sven Schewe and Bernd Finkbeiner. 2007. Bounded Synthesis. In *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4762)*, Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and

Yoshio Okamura (Eds.). Springer, 474–488. https://doi.org/10.1007/978-3-540-75596-8_33

[208] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 185:1–185:30. https://doi.org/10.1145/3360611

[209] Aravinda Prasad Sistla. 1983. *Theoretical Issues in the Design and Verification of Distributed Systems.* Ph.D. Dissertation. Harvard University.

[210] A. Prasad Sistla. 1994. Safety, Liveness and Fairness in Temporal Logic. *Formal Aspects Comput.* 6, 5 (1994), 495–512. https://doi.org/10.1007/BF01211865

[211] 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

[212] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. 1985. The Complementation Problem for Büchi Automata with Applications to Temporal Logic (Extended Abstract). In *Automata, Languages and Programming, 12th Colloquium, Nafplion, Greece, July 15-19, 1985, Proceedings.* 465–474. https://doi.org/10.1007/BFb0015772

[213] Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying k-Safety Properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 57–69. https://doi.org/10.1145/2908080.2908092

[214] Rosalie Steier. 1985. Authors. *Commun. ACM* 28, 1 (Jan. 1985), 1–2. https://doi.org/10.1145/2465.314899

[215] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021.* IEEE, 555–571. https://doi.org/10.1109/SP40001.2021.00085

[216] Dmitrii Suvorov and Vladimir Ulyantsev. 2019. Smart Contract Design Meets State Machine Synthesis: Case Studies. *CoRR* abs/1906.02906 (2019). arXiv:1906.02906 http://arxiv.org/abs/1906.02906

[217] The Anaconda Team. 2019. Understanding and Improving Conda's performance. *Anaconda Blog* (2019). Retrieved August 12, 2022 from https://www.anaconda.com/blog/understanding-and-improving-condas-performance

[218] Truffle Suite. 2021. https://www.trufflesuite.com. Accessed: 2021-11-18.

[219] Jouko Väänänen. 2008. Modal Dependence Logic. In *New Perspectives on Games and Interaction*, Vol. 4. Amsterdam University Press Amsterdam, 237–254. https://doi.org/10.5117/9789089640574

[220] Jouko A. Väänänen. 2007. *Dependence Logic - A New Approach to Independence Friendly Logic.* London Mathematical Society student texts, Vol. 70. Cambridge University Press, USA. http://www.cambridge.org/de/knowledge/isbn/item1164246/?site_locale=de_DE

[221] 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)*, Tiziana Margaria and Wang Yi (Eds.). Springer, 1–22. https://doi.org/10.1007/3-540-45319-9_1

[222] Moshe Y. Vardi. 2007. Automata-Theoretic Model Checking Revisited. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4349)*, Byron Cook and Andreas Podelski (Eds.). Springer, 137–150. https://doi.org/10.1007/978-3-540-69738-1_10

[223] Moshe Y. Vardi. 2009. From Philosophical to Industrial Logics. In *Logic and Its Applications, Third Indian Conference, ICLA 2009, Chennai, India, January 7-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5378)*, Ramaswamy Ramanujam and Sundar Sarukkai (Eds.). Springer, 89–115. https://doi.org/10.1007/978-3-540-92701-3_7

[224] 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

[225] Jonni Virtema, Jana Hofmann, Bernd Finkbeiner, Juha Kontinen, and Fan Yang. 2021. Linear-Time Temporal Logic with Team Semantics: Expressivity and Complexity. In *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2021, December 15-17, 2021, Virtual Conference (LIPIcs, Vol. 213)*, Mikolaj Bojanczyk and Chandra Chekuri

(Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 52:1–52:17. https://doi.org/10.4230/LIPIcs.FSTTCS.2021.52

[226] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2021. oo7: Low-Overhead Defense Against Spectre Attacks via Program Analysis. *IEEE Trans. Software Eng.* 47, 11 (2021), 2504–2519. https://doi.org/10.1109/TSE.2019.2953709

[227] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2019. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12031)*, Supratik Chakraborty and Jorge A. Navas (Eds.). Springer, 87–106. https://doi.org/10.1007/978-3-030-41600-3_7

[228] Yu Wang, Siddhartha Nalluri, Borzoo Bonakdarpour, and Miroslav Pajic. 2021. Statistical Model Checking for Hyperproperties. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–16. https://doi.org/10.1109/CSF51468.2021.00009

[229] Microsoft Azure Blockchain Workbench. 2019. *Asset Transfer Contract*. Retrieved August 12, 2022 from https://github.com/Azure-Samples/blockchain/blob/master/blockchain-workbench/application-and-smart-contract-samples/asset-transfer/ethereum/AssetTransfer.sol

[230] Microsoft Azure Blockchain Workbench. 2021. *Asset transfer sample from the Azure Blockchain Workbench*. Retrieved August 12, 2022 from https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/asset-transfer

[231] Fan Yang and Jouko Väänänen. 2016. Propositional Logics of Dependence. *Annals of Pure and Applied Logic* 167, 7 (2016), 557 – 589. https://doi.org/10.1016/j.apal.2016.03.003

[232] Fan Yang and Jouko Väänänen. 2017. Propositional Team Logics. *Annals of Pure and Applied Logic* 168, 7 (2017), 1406 – 1441. https://doi.org/10.1016/j.apal.2017.01.007

[233] 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*. 29. https://doi.org/10.1109/CSFW.2003.1212703

[234] Lantian Zheng and Andrew C. Myers. 2004. Dynamic Security Labels and Noninterference (Extended Abstract). In *Formal Aspects in Security and Trust: Second IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST), an event of the 18th IFIP World Computer Congress, August 22-27, 2004, Toulouse, France (IFIP, Vol. 173)*, Theodosis Dimitrakos and Fabio Martinelli (Eds.). Springer, 27–40. https://doi.org/10.1007/0-387-24098-5_3

[235] Nejc Zupan, Prabhakaran Kasinathan, Jorge Cuellar, and Markus Sauer. 2020. Secure Smart Contract Generation Based on Petri Nets. In *Blockchain Technology for Industry 4.0*. Springer, Singapore, 73–98. https://doi.org/10.1007/978-981-15-1137-0_4